Exploiting Short Supports for Generalised Arc Consistency for Arbitrary Constraints

Peter Nightingale, Ian P. Gent, Chris Jefferson, Ian Miguel
School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK
{pn,ipg,caj,ianm}@cs.st-andrews.ac.uk

Abstract

Special-purpose constraint propagation algorithms (such as those for the element constraint) frequently make implicit use of *short supports* — by examining a subset of the variables, they can infer support for all other variables and values and save substantial work. However, to date general purpose propagation algorithms (such as GAC-Schema) rely upon supports involving all variables. We demonstrate how to employ short supports in a new general purpose propagation algorithm called SHORTGAC. This works when provided with either an explicit list of allowed short tuples, or a function to calculate the next supporting short tuple. Empirical analyses demonstrate the efficiency of SHORTGAC compared to other general-purpose propagation algorithms. In some cases SHORTGAC even exhibits similar performance to special-purpose propagators.

1 Introduction

Constraint solving of a given problem proceeds in two phases. First, the problem is *modelled* as a set of decision variables and 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, and its associated domain of potential values corresponds to the options for that choice. In the second phase, a constraint solver searches for a solution or solutions automatically. Typically, constraint solvers employ a systematic backtracking search, interleaving the choice of an instantiation of a decision variable with the *propagation* of the constraints to determine the consequences of the choice made. In order to facilitate this process, each constraint supported by the solver has an associated *propagation algorithm*.

Propagation algorithms can broadly be divided into two types. The first are specialised to reason very efficiently about constraint patterns that occur very frequently in models. Examples include the global cardinality constraint [Régin, 1996] and the element constraint [Gent *et al.*, 2006b]. It is not feasible to support every possible constraint expression with a specialised propagator in this way, in which case general-purpose constraint propagators, such as GAC-Schema [Bessière and Régin, 1997] or GAC2001 [Bessière *et al.*, 2005], are used. These are typically more expensive than specialised propagators but are an important tool when no specialised propagator is available.

A *support* for a domain value of a variable is an explanation as to why that value cannot yet be removed from consideration in the search for a solution. It is usually given in terms of a set of *literals*: variable-value pairs corresponding to possible assignments to the other variables in the constraint. One of the efficiencies typically found in specialised propagators is the (typically implicit) use of *short supports*: by examining a subset of the variables, they can infer support for all other variables and values and save substantial work.

For example, consider the $\mathrm{element}([x_0, x_1, x_2], y, z)$ constraint, with $x_0, x_1, x_2, y \in \{0 \dots 2\}$, $z \in \{0 \dots 3\}$. The constraint is satisfied iff $x_y = z$ (i.e. the element in position y of vector X equals z). Consider the set of literals $S_1 = \{x_0 \mapsto 1, y \mapsto 0, z \mapsto 1\}$. This set clearly satisfies the definition of the constraint $x_y = z$, but it does not contain a literal for each variable. Any extension of S with valid literals for variables x_1 and x_2 is a support. We call S a *short support*. As a second example, the watched literal propagation algorithm in SAT exploits short supports containing one variable, requiring only two such supports for a constraint of any length.

To date, general-purpose propagation algorithms rely upon supports involving all variables. In this paper, we demonstrate how to employ short supports in a new general-purpose propagation algorithm called SHORTGAC. As we will demonstrate, the use of short supports significantly improves the performance of SHORTGAC versus existing general-purpose propagation algorithms. In some cases, SHORTGAC even approaches the performance of special-purpose propagators.

2 Supports, Short Supports and GAC

A literal $x \mapsto v$ is *valid* if v is in the current domain of x. The *scope* of a constraint c is the subset of variables that it constrains. A *valid tuple* for constraint c is a set of literals that contains a valid literal for each variable in the scope of c and satisfies c. With respect to a constraint c, a *support* for a literal l is a valid tuple for c that contains l.

Each *short* support is a set of literals that contains a valid literal for each of some *subset* of variables in the scope of c, and such that *every* superset of this containing one valid literal for each variable in c is a support. For example, even the empty set is a possible short support, if every remaining tuple of valid literals satisfies c (in which case c is 'entailed'.)

The property commonly established by constraint propagation algorithms is *generalised arc consistency* (GAC) [Mackworth, 1977]. A constraint c is GAC if and only if there exists a support for every valid literal of every variable in the scope of c. GAC is established by identifying all literals $x \mapsto v$ for which no support exists and removing v from the domain of x.

A GAC propagation algorithm is usually situated in a systematic search. Hence, it must operate in three contexts: initialisation (at the root node), where support is established from scratch; following the deletion of one or more domain values (as a result of a branching decision and/or the propagation of other constraints), where support must be re-established selectively; and upon backtracking, where data structures must be restored to the correct state for this point in search. Our primary focus will be on the second context, operation following value deletion, although we will discuss the other two briefly.

A GAC propagation algorithm would typically be called for each deleted domain value in turn. Once the algorithm has been called for each such domain value, the constraint will be GAC. However, the propagation algorithm proposed here need not be called for *every* deleted domain value to establish GAC. In this paper we say that propagators attach and remove *triggers* on literals. When a domain value v for variable x is deleted, the propagator is called if and only if it has a trigger attached to the literal $x \mapsto v$.

3 SHORTGAC: An Overview

This section summarises the key ideas of the SHORTGAC propagation algorithm, along with an illustrative example.

SHORTGAC maintains a set of short supports sufficient to support all valid literals of the variables in the scope of the constraint it is propagating. We refer to these as the *active* supports. The major contribution of the algorithm is the efficient storage and update of the set of active supports. This efficiency rests on exploiting the observation that, using short supports, support can be established for a literal in two ways. First, as usual, a short support that contains a literal supports that literal. Second, a literal $x \mapsto v$ is supported by a short support that contains no literal of variable x. Hence, the only short supports that do *not* support $x \mapsto v$ are those which contain a literal $x \mapsto w$ for some other value $w \neq v$.

The following counters are central to the operation of the SHORTGAC algorithm:

numSupports is the total number of active supports.

supportsPerVar$[x]$ is an array indicating the number of supports containing each variable x.

supportsPerLit$[x, v]$ is a 2-dimensional array indicating the number of supports containing each literal $x \mapsto v$.

If the number of supports containing some variable x is less than the total number of supports then there exists a support s that does not contain x. Therefore, as noted, s supports all literals of x. The algorithm spends no time processing variables all of whose literals are known to be supported in this way. Only for variables involved in all supporting tuples do we have to seek support for literals with no current supports.

To illustrate, we consider the element example. Suppose in the current state SHORTGAC is storing just one support: $S_1 = \{x_0 \mapsto 1, y \mapsto 0, z \mapsto 1\}$. The counters are as follows. (X indicates that a literal is not valid.)

supportsPerLit:	Variable				
Value	x_0	x_1	x_2	y	z
0	0	0	0	1	0
1	1	0	0	0	1
2	0	0	0	0	0
3	X	X	X	X	0
supportsPerVar:	1	0	0	1	1
numSupports:	1				

All values of x_1 and x_2 have support, since their supportsPerVar counters are both less than numSupports. Therefore the SHORTGAC algorithm can ignore x_1 and x_2 and only look for new supports of x_0, y and z. Consider finding a new support for literals in z. SHORTGAC can ignore the literals with at least one support – in this case $z \mapsto 1$. The algorithm looks for literals $z \mapsto a$ where supportsPerLit[z, a] = 0. Here, $z \mapsto 0$ is such a literal, so SHORTGAC seeks a new support for it. A possible new support is $S_2 = \{x_1 \mapsto 0, y \mapsto 1, z \mapsto 0\}$, with which the counters are updated:

supportsPerLit:	Variable				
Value	x_0	x_1	x_2	y	z
0	0	1	0	1	1
1	1	0	0	1	1
2	0	0	0	0	0
3	X	X	X	X	0
supportsPerVar:	1	1	0	2	2
numSupports:	2				

Now variable x_0 is also fully supported, since supportsPerVar[x_0] < numSupports. However, we retain the values of supportsPerLit for x_0 in case numSupports falls back to 1 subsequently. There remain three literals for which support has not been established: $y \mapsto 2, z \mapsto 2$ and $z \mapsto 3$. For the first two SHORTGAC finds supports such as $S_3 = \{x_0 \mapsto 2, y \mapsto 0, z \mapsto 2\}$, $S_4 = \{x_2 \mapsto 0, y \mapsto 2, z \mapsto 0\}$. No support exists for $z \mapsto 3$, so this literal is invalid and 3 will be deleted, giving:

supportsPerLit:	Variable				
Value	x_0	x_1	x_2	y	z
0	0	1	1	2	2
1	1	0	0	1	1
2	1	0	0	1	1
3	X	X	X	X	X
supportsPerVar:	2	1	1	4	4
numSupports:	4				

All valid literals are now supported. Nothing further need be done until a change in state, such as the removal of a value by propagation, or a change in state following backtracking.

4 SHORTGAC: Details

The key tasks are: counter update; iteration over variables where supportsPerVar equals numSupports; and iteration over the unsupported values of a variable. The following data structures allow us to do each of these tasks efficiently.

A short support is represented by a struct. It may be contained in multiple doubly-linked lists simultaneously (one for each literal $x \mapsto v$ in the support). Hence, it has multiple previous and next pointers. The support struct contains:

prev$[x]$ An array of previous pointers, indexed by variable.
next$[x]$ An array of next pointers, indexed by variable.
literals An array of literals

The doubly-linked lists are accessed via **supportListPerLit**$[x, v]$, an array of pointers to supports. If an active support S contains a literal $x \mapsto v$, then S is contained in the doubly-linked list supportListPerLit$[x, v]$. Hence, it is possible to iterate through all supports containing any particular literal.

The algorithm iterates over all variables x where supportsPerVar$[x]$ equals numSupports. The following data structure represents a partition of the variables by the number of supports. It allows constant time size checking and linear-time iteration of each cell in the partition, and allows any variable to be moved into an adjacent cell (*ie* if the number of supports increases or decreases by 1) in constant time. It is inspired by the indexed dependency array in [Schulte and Tack, 2010].

varsPerSupport is an array containing a permutation of the variables. Variables are ordered by their number of supports (supportsPerVar$[x]$) in ascending order.
varsPerSupInv is the inverse mapping of varsPerSupport.
supportNumPtrs is an array of integers such that supportNumPtrs$[i]$ is the smallest index in varsPerSupport with i or more supports. Therefore varsPerSupport[supportNumPtrs$[i]$..supportNumPtrs$[i+1]$-1] contains exactly the set of variables with i supports.

Procedure swap(x_i, x_j) (used later) locates and swaps the two variables in varsPerSupport, also updating varsPerSupInv.

For a variable x with supportsPerVar$[x]$ = numSupports, SHORTGAC iterates over the values with zero supports. To avoid iterating over all values, we use a set data structure:

zeroVals$[x]$ is a stack containing the zero values for x in no particular order.
inZeroVals$[x, v]$ is a Boolean indicating whether value v is on the stack.

When supportsPerLit$[x,v]$ is reduced to 0, if inZeroVals$[x,v]$ is false then v is pushed onto zeroVals$[x]$. As an optimisation, values are not eagerly removed from the set; they are only removed when the set is iterated. Also, the set is not backtracked. During iteration, a non-zero value is removed by swapping it to the top of the stack, and popping.

4.1 Adding and Deleting Supports

When a support is added or deleted, all the data structures described above are updated. This is done by the addSupport and deleteSupport procedures. AddSupport is given in Algorithm 1. For each literal in the support *sup*, *sup* is added to the appropriate supportListPerLit, and the two counters supportsPerVar and supportsPerLit are incremented. x_i must also be moved to the next cell in varsPerSupport. Line 6 finds the end of the cell that x_i is in. Line 7 swaps x_i to the end of its cell, and line 8 moves a cell boundary such that x_i is now in the higher cell. Lines 9-10 add a trigger to $x_i \mapsto a$ if *sup* is the only active support to contain that literal (otherwise $x_i \mapsto a$ will already have a trigger).

The procedure deleteSupport is similar with the following changes. On lines 3, 4 and 11, counters are decremented.

Algorithm 1 Add Support: addSupport(*sup*)

```
Require: sup: a support struct
 1: for all $(x_i \mapsto a)$ in sup.literals do
 2:     Add sup to supportListPerLit[$x_i$, $a$]
 3:     supportsPerVar[$x_i$]← supportsPerVar[$x_i$]+1
 4:     supportsPerLit[$x_i$,$a$]← supportsPerLit[$x_i$,$a$]+1
 5:     sPV←supportsPerVar[$x_i$]
 6:     cellend ←supportNumPtrs[sPV]$-1$
 7:     swap($x_i$, varsPerSupport[cellend])
 8:     supportNumPtrs[sPV]←supportNumPtrs[sPV]$-1$
 9:     if supportsPerLit[$x_i$,$a$]=1 then
10:         attachTrigger($x_i$,$a$)
11: numSupports←numSupports+1
```

On line 2, *sup* is removed from the list (in constant time). Lines 6-8 are altered to move x_i into the lower adjacent cell in varsPerSupport. Line 6 finds the start of the cell that x_i is in (supportNumPtrs[sPV+1]), line 7 remains the same and line 8 increments the lower boundary (supportNumPtrs[sPV+1]) so that x_i is now in the adjacent cell. Lines 9-10 call removeTrigger if supportsPerLit$[x_i,a]$=0. Between lines 8 and 9, a new section is added: if supportsPerLit$[x_i,a]$=0, then value a is added to zeroVals$[x_i]$ if not already present.

All changes to data structures (except where noted above) are stored on a stack; addSupport and deleteSupport are used to revert changes upon backtracking.

4.2 The Propagation Algorithm

The SHORTGAC propagator (Algorithm 2) is only invoked when a literal contained in one or more active supports is pruned. It first calls deleteSupport for all active supports containing the pruned literal. Then it iterates through all variables x_j where supportsPerVar$[x_j]$=numSupports (lines 3-4). x_j may have unsupported values. SHORTGAC iterates through zeroVals$[x_j]$, discarding values with support (lines 6-7), and seeking a new support for those without (lines 10-16).

A new support is sought by calling findNewSupport(x_j, a). This returns a support for the literal if one exists. If there is no support, the literal is pruned. If there exists a support (sup) it is added on line 14. sup must support $x_j \mapsto a$, but it does not have to contain $x_j \mapsto a$. a is removed from zeroVals$[x_j]$ only if sup contains $x_j \mapsto a$. At this point, numSupports and supportNumPtrs have changed, so the loop (lines 3-17) is restarted (line 17) using a goto statement. The aim is to process all variables where supportsPerVar$[x_j]$=numSupports; adding a new support changes this set of variables, therefore the loop is restarted.

Complexity analysis of the algorithm is left for future work, and we do not claim optimality. In particular the algorithm can process variables where each value is supported. To initialise the propagation process, lines 3-17 of Algorithm 2 are invoked.

4.3 Instantiation of findNewSupport

Similarly to GAC-Schema [Bessière and Régin, 1997], SHORTGAC must be instantiated with a findNewSupport function. The function takes a valid literal, and returns a support if one exists, otherwise returns null. A findNewSupport function

Algorithm 2 SHORTGAC-Propagate: propagate(var, val)

```
Require: $var$, $val$ (where $val$ has been pruned from $var$)
 1: while supportListPerLit[$var$, $val$] $\neq$ null do
 2:   deleteSupport(supportListPerLit[$var$, $val$])
 3: for all $i \in \{$supportNumPtrs[numSupports]...
                supportNumPtrs[numSupports+1]-1$\}$ do
 4:   $x_j \leftarrow$ varsPerSupport[$i$]
 5:   for all $a \in \mathrm{zeroVals}[x_j]$ do
 6:     if $\mathrm{supportsPerLit}[x_j, a] > 0$ then
 7:       Remove $a$ from zeroVals[$x_j$]
 8:     else
 9:       if $a \in \mathrm{Dom}(x_j)$ then
10:         sup$\leftarrow$findNewSupport($x_j, a$)
11:         if sup=null then
12:           prune($x_j$,$a$)
13:         else
14:           addSupport(sup)
15:           if $\mathrm{supportsPerLit}[x_j, a] > 0$ then
16:             Remove $a$ from zeroVals[$x_j$]
17:           Goto 3
```

may be written for a particular constraint. In the case studies below we briefly summarise what this function does.

Alternatively, we provide a generic instantiation named findNewSupport-List (Algorithm 3) that takes a list of short supports for each literal (supportList). This is analogous to the Positive instantiation of GAC-Schema [Bessière and Régin, 1997]. FindNewSupport-List has persistent state: listPos, an array of integers indexed by variable and value, initially 0. This indicates the current position in the supportList. ListPos is not backtracked. The algorithm simply iterates through the list of supports, seeking one where all literals are valid. Using listPos stops the algorithm repeatedly iterating through the first section of the list when called repeatedly. Note that a constraint-specific findNewSupport can sometimes find shorter supports than findNewSupport-list. This is because a specific findNewSupport can take advantage of current domains whereas the supportList may only contain supports given the initial domains. For example, if the constraint becomes entailed, the specific findNewSupport can return the empty support. We exploit this fact in case study 3 below.

As an optimisation, we omit literals from sup for assigned variables. This is applied to all findNewSupport functions (except for SHORTGAC-longsupport, described below).

5 Experimental Setup

The Minion solver 0.10 [Gent *et al.*, 2006a] was used for experiments, with our own additions. We used an 8-core machine with 2.27GHz Intel Xeon E5520 CPUs and 12GB memory. All times are a median of 5 runs. We report complete run times, i.e. we have not attempted to measure the time attributable to the given propagator. This both simplifies our experiments and has the advantage that we automatically take account of all factors affecting runtime.

For each case study, we implemented a findNewSupport method for SHORTGAC specific to the constraint. We also used the generic list instantiation for comparison where possible. We compare SHORTGAC against the special-purpose propagator (when available). We also compare against SHORTGAC-longsupport (SHORTGAC with full length supports), and against GAC-Schema [Bessière and Régin, 1997] as the closest equivalent algorithm without short supports. Our implementation of GAC-Schema is very similar to SHORTGAC, using the same data structures where possible. GAC-Schema and SHORTGAC-longsupport use the same (constraint-specific) findNewSupport as SHORTGAC, and subsequently extend the short support to full length using the minimum value for each extra variable. In each case, the constraint can be compactly represented as a disjunction. Therefore we compare SHORTGAC against Constructive Disjunction. The algorithm used is based upon [Lagerkvist and Schulte, 2009], without entailment detection. We do not compare to table constraints (*eg* [Gent *et al.*, 2007]) because the constraints are too large. For example, the smallest element constraints reported below have 6^{38} valid tuples, making it impossible even to generate and store the list of allowed tuples.

6 Case Study 1: Element

We use the quasigroup existence problem QG3 [Colton and Miguel, 2001] to evaluate SHORTGAC on the element constraint. The problem class has one parameter n, specifying the size of an $n \times n$ table (qg) of variables with domains $\{0 \dots n\}$. Rows, columns and one diagonal have GAC allDifferent constraints, following Colton and Miguel's model. The element constraints represent the QG3 property that $(i * j) * (j * i) = i$. This translates as $\forall i, j$: element(qg, aux[i, j], i), and aux[i, j]= $n \times qg[i, j] + qg[j, i]$, where aux[$i, j$] has domain $\{0 \dots n \times n - 1\}$.

For the constraint element(X, y, z), the findNewSupport method for SHORTGAC returns tuples of the form $\langle x_i \mapsto j, y \mapsto i, z \mapsto j \rangle$, where i is an index into the vector X and j is a common value of z and x_i. SHORTGAC-list has all supports of this form. For Constructive Or, we used $(x_0 = z \wedge y = 0) \vee \cdots \vee (x_n = z \wedge y = n)$.

We searched for all solutions, with limits of 100,000 nodes, 1,800 seconds, and 12GB of virtual memory. Table 1 shows our results on QG3. Of the general purpose methods, using short supports in either functional or list form is dramatically better than any alternative. For example at $n = 10$, even the

Algorithm 3 findNewSupport-list: findNewSupport(x_i, a)

```
Require: $x_i$, $a$, supportList
 1: for all $j \in \{$listPos[$x_i$, $a$]...(supportList[$x_i$, $a$].size-1)$\}$
    do
 2:   $sup$ $\leftarrow$supportList[$x_i$, $a$, $j$]
 3:   if all literals in $sup$ are valid then
 4:     listPos[$x_i$, $a$]$\leftarrow$ $j$
 5:     return $sup$
 6: for all $j \in \{0 \dots$listPos[$x_i$, $a$]$-1\}$ do
 7:   $sup$ $\leftarrow$supportList[$x_i$, $a$, $j$]
 8:   if all literals in $sup$ are valid then
 9:     listPos[$x_i$, $a$]$\leftarrow$ $j$
10:     return $sup$
11: return null
```

n	WatchElt	SG	SG-L	GAC-S	SG-Ls	Or
6	22260	4839	2100	25.2	11.0	68.1
7	22731	3736	2539	8.83	3.29	34.2
8	16287	2238	1388	3.78	1.26	13.1
9	16129	1961	1115	2.18	0.756	8.27
10	18939	2149	1088	mem	0.373	5.45

Table 1: Nodes searched per second for quasigroup existence problems. 'mem' indicates running out of memory. Columns are special purpose Watched Element propagator (WatchElt), SHORTGAC (SG), SHORTGAC-list (SG-L), GAC-Schema (GAC-S), SHORTGAC-longsupport (SG-Ls), and Constructive Or (Or)

slower, list based method is 200 times faster than Constructive Or, the best of the other methods. Functional SHORTGAC is about twice as fast as SHORTGAC-list, but cannot compete with the special purpose element propagator in Minion [Gent *et al.*, 2006b] which is about 5-9 times faster. It remains clear that exploiting short supports is very beneficial compared to other general purpose methods.

7 Case Study 2: Lex-ordering

We use BIBD problems to evaluate SHORTGAC on the lexicographic ordering constraint. The lex constraint is placed on both the rows and columns, to perform the "Double Lex" symmetry breaking method. We use the BIBD model and GACLex propagator given by [Frisch *et al.*, 2002]. We use BIBDs with the parameter values $(4n+3, 4n+3, 2n+1, 2n+1, n)$ and we impose a 2GB limit on runs. All methods explore identical search spaces. We searched for all solutions up to a node limit of 1,000,000 or time limit of 1800s (whichever was first). This limits all instances except $n = 3$ which required 41,982 nodes.

For the constraint lexleq(X, Y), we define $mx_i = min(Dom(x_i))$ and $my_i = max(Dom(y_i))$. The findNewSupport method for SHORTGAC finds the lowest index $i \in \{0 \dots n\}$ such that $mx_i < my_i$, or $i = n$. The case $i = n$ arises when X cannot be lex-less than Y, so a support is sought for $X = Y$. If $i < n$, the support contains $x_i \mapsto mx_i$, $y_i \mapsto my_i$. For each index $j < i$, if $mx_j = my_j$, then the short support contains $x_j \mapsto mx_j, y_j \mapsto my_j$ otherwise there is no valid support and null is returned.

The lex constraint on two arrays of length n and domain size d has more than d^n short supports since all assignments where the two arrays are equal satisfy the constraint and cannot be reduced. SHORTGAC-list is not practical for any substantial constraint. Constructive Or uses the following representation: $(x_0 < y_0) \vee (x_0 = y_0 \wedge x_1 < y_1) \vee \cdots$, including the case where all pairs are equal.

Table 2 shows the results of our experiments on non-list based methods with values of n from 3 to 24. It is clear that the best method is the special purpose Lex propagator, up to twice as fast as SHORTGAC. The speedup of Lex over SHORTGAC increases only very slowly, from 1.47 at $n = 3$ to 2.02 at $n = 24$. SHORTGAC is by far the best general purpose method. At $n = 3$ SHORTGAC is 6.9 times better than the best alternative general-purpose method, and by $n = 24$ SHORTGAC is 692 times faster.

n	Lex	SG	GAC-S	SG-Ls	Or
3	104955	71156	3540	3103	10265
4	113379	88731	3555	2644	7243
5	97371	82645	3077	2297	6035
6	81301	70671	1766	1413	3276
7	71276	62617	1438	1015	2251
8	66756	48497	783	650	1197
9	62854	43764	510	383	504
10	56657	36232	315	304	281
12	48426	29070	143	130	145
14	37341	21409	78.1	69.9	97.0
16	31949	16795	57.5	47.7	mem
18	24564	13208	34.8	28.6	mem
20	19342	10453	17.7	12.3	mem
22	15354	8078	11.2	9.51	mem
24	12228	6046	8.74	7.06	mem

Table 2: Results for BIBDs. We report nodes searched per second, with a limit of the first of either 1,000,000 nodes or 1,800s. Lex indicates GACLex, and other titles as in Table 1.

n-w-h	SG	SG-L	GAC-S	SG-Ls	Or
18-31-69	5.00	23.96	35.58	119.44	121.01
19-47-53	1.77	20.72	33.52	74.30	109.86
20-34-85	6.72	41.98	50.31	209.59	206.22
21-38-88	8.45	43.63	56.54	222.84	208.17
22-39-98	9.32	58.85	66.91	276.69	284.70
23-64-68	1.93	46.80	57.52	121.38	145.22
24-56-88	8.20	76.83	74.51	232.56	352.77
25-43-129	16.98	mem	101.10	519.73	526.88
26-70-89	2.33	mem	84.95	231.02	197.51
27-47-148	25.41	mem	128.22	764.74	699.66

Table 3: Times (seconds) for rectangle packing. Column titles as in Table 1.

8 Case Study 3: Rectangle Packing

The rectangle packing problem [Simonis and O'Sullivan, 2008] (with parameters n, *width* and *height*) consists of packing all squares from size 1×1 to $n \times n$ into the rectangle of size $width \times height$. This is modelled as follows: we have variables $x_1 \dots x_n$ and $y_1 \dots y_n$, where (x_i, y_i) represents the cartesian coordinates of the lower-left corner of the $i \times i$ square. Domains of x_i variables are $\{0 \dots width - i\}$, and for y_i variables are $\{0 \dots height - i\}$. Variables are branched in decreasing order of i, with x_i before y_i, smallest value first. The only type of constraint is non-overlap of squares i and j: $(x_i + i \leq x_j) \vee (x_j + j \leq x_i) \vee (y_i + i \leq y_j) \vee (y_j + j \leq y_i)$ Minion does not have the special-purpose non-overlap constraint [Simonis and O'Sullivan, 2008], so we only report a comparison of general purpose methods.

The domains of x_n and y_n are reduced to break flip symmetries as described in [Simonis and O'Sullivan, 2008]. Our focus is performance of the non-overlap constraint, and so we did not implement the commonly-used implied constraints.

The findNewSupport function for SHORTGAC is as follows. If any of the four disjuncts above are entailed given the current domains, return the empty support (indicating entailment). Otherwise, return a support with 2 literals to satisfy one of the four disjuncts. SHORTGAC-list has all supports of size 2.

For the experiment we used the optimum rectangle sizes reported by [Simonis and O'Sullivan, 2008]. Virtual memory was limited to 12GB. Times are given for the first 50,000 nodes of search in Table 3. We can see that SHORTGAC is by far the best technique. Of the other methods, GAC-Schema is second-best, but is always at least 5 times slower and can be almost 40 times slower. SHORTGAC-list is faster than GAC-Schema until it reaches larger problems, when it is first slightly slower and then runs out of memory. GAC-Schema in turn is significantly faster than either using SHORTGAC with full length tuples, or using Constructive Or. In summary, these results very clearly show the benefits of using short supports.

9 Related Work

Our use of counters to count supports is inspired by AC4 [Mohr and Henderson, 1986], and is also related to the concept of quantitative supports [Jain *et al.*, 2010]. There has been study of compressing the tuples of a constraint (*eg* using tries [Gent *et al.*, 2007]). Such techniques are orthogonal to this paper because they only assist in finding full-length supports.

Katsirelos and Walsh [2007] proposed a different generalisation of support, called a c-tuple. A c-tuple is a set of literals, such that every valid tuple contained in the c-tuple satisfies the constraint. Katsirelos and Walsh give an outline of a modified version of GAC-Schema which directly stores c-tuples. While c-tuples offer space savings when storing constraints, Katsirelos and Walsh found using c-tuples provides almost no performance improvement over GAC-Schema.

For Constructive Or, Lhomme observed that a support for one disjunct A will support all values of any variable not contained in A. The concept is similar to a short support albeit less general, because the length of the supports is fixed to the length of the disjuncts. He presents a non-incremental constructive disjunction for two disjuncts [Lhomme, 2003].

10 Conclusions

We have introduced and described in detail SHORTGAC, a general purpose propagation algorithm for short supports. Either it can be given a specialised function to find new supports for each constraint, or used with a method which accepts an explicit list of short supports. In three case studies, SHORTGAC is far faster than general purpose methods. In the best case it achieved speeds about 50% of that of a special purpose propagator. We conclude that where short supports are available, SHORTGAC will provide much better performance than existing general-purpose constraint propagation methods.

We identify two areas of interest for future work. First, we observed that SHORTGAC with lists has memory problems. Providing a practically effective version of the table constraint with short supports would further increase the utility of our work. Second, we see that SHORTGAC with full length supports was significantly slower than GAC-Schema. It would be beneficial if a single propagation algorithm could be found that was as fast as SHORTGAC on short supports and as fast as GAC-Schema on long supports.

Acknowledgements
We would like to thank anonymous reviewers for their comments, and EPSRC for funding this work through grants EP/H004092/1 and EP/E030394/1.

References

[Bessière and Régin, 1997] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proc. IJCAI 97*, pages 398–404, 1997.

[Bessière *et al.*, 2005] C. Bessière, J.-C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[Colton and Miguel, 2001] S. Colton and I. Miguel. Constraint generation via automated theory formation. In *Proc. CP 2001*, pages 575–579, 2001.

[Frisch *et al.*, 2002] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. CP 2002*, pages 93–108, 2002.

[Gent *et al.*, 2006a] I. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. In *Proc. ECAI 2006*, pages 98–102, 2006.

[Gent *et al.*, 2006b] I. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Proc. CP 2006*, 2006.

[Gent *et al.*, 2007] I. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proc. AAAI-07*, pages 191–197, 2007.

[Jain *et al.*, 2010] S. Jain, E. O'Mahony, and M. Sellmann. A complete multi-valued sat solver. In *Proc. CP 2010*, pages 281–296, 2010.

[Katsirelos and Walsh, 2007] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proc. CP 2007*, pages 379–393, 2007.

[Lagerkvist and Schulte, 2009] M. Lagerkvist and C. Schulte. Propagator groups. In *Proc. CP 2009*, pages 524–538, 2009.

[Lhomme, 2003] O. Lhomme. An efficient filtering algorithm for disjunction of constraints. In *Proc. CP 2003*, pages 904–908, 2003.

[Mackworth, 1977] A. K. Mackworth. On reading sketch maps. In *Proc. IJCAI 77*, pages 598–606, 1977.

[Mohr and Henderson, 1986] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[Régin, 1996] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. AAAI 96*, pages 209–215, 1996.

[Schulte and Tack, 2010] C. Schulte and G. Tack. Implementing efficient propagation control. In *Proc. TRICS 2010*, 2010.

[Simonis and O'Sullivan, 2008] H. Simonis and B. O'Sullivan. Search strategies for rectangle packing. In *Proc. CP 2008*, pages 52–66, 2008.