Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Extending Simple Tabular Reduction with Short Supports

Christopher Jefferson, Peter Nightingale
School of Computer Science, University of St Andrews
St Andrews, Fife KY16 9SX, UK
{caj21,pwn1}@st-andrews.ac.uk

Abstract

Constraint propagation is one of the key techniques in constraint programming, and a large body of work has built up around it. Special-purpose constraint propagation algorithms frequently make implicit use of *short supports* — by examining a subset of the variables, they can infer support (a justification that a variable-value pair still forms part of a solution to the constraint) for all other variables and values and save substantial work. Recently short supports have been used in general purpose propagators, and (when the constraint is amenable to short supports) speed ups of more than three orders of magnitude have been demonstrated.

In this paper we present SHORTSTR2, a development of the Simple Tabular Reduction algorithm STR2+. We show that SHORTSTR2 is complementary to the existing algorithms SHORTGAC and HAGGISGAC that exploit short supports, while being much simpler. When a constraint is amenable to short supports, the short support set can be exponentially smaller than the full-length support set. Therefore SHORTSTR2 can efficiently propagate many constraints that STR2+ cannot even load into memory. We also show that SHORTSTR2 can be combined with a simple algorithm to identify short supports from full-length supports, to provide a superior drop-in replacement for STR2+.

1 Introduction

Constraint solvers typically employ a systematic backtracking search, interleaving the choice of an assignment of a decision variable with the *propagation* of the constraints to determine the consequences of the assignment made. Propagation algorithms can broadly be divided into two types. The first are specialised to reason very efficiently about constraint patterns that occur frequently in models, for example the element propagator [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], GAC2001/3.1 [Bessière *et al.*, 2005], STR2 [Lecoutre, 2011] or MDDC [Cheng and Yap, 2010] are used. These are typically more expensive than specialised propagators but are an important tool nonetheless.

A *support* in a constraint for a domain value of a variable is a justification that the value still forms part of an assignment that satisfies the constraint. It is usually given in terms of a set of *literals*: variable-value pairs that are possible assignments to the variables in the constraint. One efficiency measure often found in specialised propagators is *short supports*: by examining a subset of the variables, they can infer support for all other variables and values and save substantial work. Short supports are typically implicit in a specialised algorithm that does not examine all variables in all cases. Short support is an important concept for general-purpose propagation [Nightingale *et al.*, 2011; 2013].

Consider the element constraint $x_y = z$, with variables $x_0, x_1, x_2, y \in \{0 \dots 2\}, z \in \{0 \dots 3\}$. This constraint is satisfied iff the element in position y of vector $[x_0, x_1, x_2]$ equals z. Consider the set of literals $S = \{x_0 \mapsto 1, y \mapsto 0, z \mapsto 1\}$. 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 valid support. S is an example of a *short support*.

The algorithms SHORTGAC [Nightingale *et al.*, 2011], HAGGISGAC and HAGGISGAC-STABLE [Nightingale *et al.*, 2013] are based on the classic general-purpose GAC algorithm GAC-Schema [Bessière and Régin, 1997]. Compared to GAC-Schema, these algorithms exhibited orders of magnitude speed improvements when the constraint is amenable to using short supports. The later algorithms HAGGISGAC and HAGGISGAC-STABLE are also well-behaved with full-length supports, performing better than GAC-Schema.

In this paper we take a different approach of building on the Simple Tabular Reduction algorithm STR2+. Since STR2+ has different strengths and weaknesses to GAC-Schema, we expect that the resulting SHORTSTR2 (i.e. STR2+ exploiting short supports) will be entirely different and possibly complementary to SHORTGAC, HAGGISGAC and HAGGISGAC-STABLE. Also, SHORTSTR2 will be a much simpler algorithm and therefore more attractive to implement in solvers.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined as a set of variables X, a function that maps each variable to its domain,

$D : X \rightarrow 2^{\mathbb{Z}}$ where each domain is a finite set, and a set of constraints C. A constraint $c \in C$ is a relation over a subset of the variables X. The *scope* of a constraint c, named $\text{scope}(c)$, is the set of variables that c constrains.

During a systematic search for a solution to a CSP, values are progressively removed from the domains D. Therefore, we distinguish between the *initial* domains and the *current* domains. The function D refers to the current domains. A *literal* is a variable-value pair (written $x \mapsto v$). A literal $x \mapsto v$ is *valid* if $v \in D(x)$. The size of the largest initial domain is d. For a constraint c we use r for the size of $\text{scope}(c)$.

A *full-length support* of constraint c is a set of literals containing exactly one literal for each variable in $\text{scope}(c)$, such that c is satisfied by the assignment represented by these literals. Following [Nightingale *et al.*, 2013] we define short support as follows.

Definition 1. *[Short support] A **short support** S for constraint c and domains D_s is a set of literals $x \mapsto v$ such that $x \in \text{scope}(c)$, $x \mapsto v$ is valid w.r.t D_s, x occurs only once in S, and every superset of S that contains one valid (w.r.t D_s) literal for each variable in $\text{scope}(c)$ is a full-length support.*

The set of short supports depends on the domains D_s. In this paper we always use the initial domains for D_s. Elsewhere, short supports are generated using the current domains D but these sets are not necessarily short supports after backtracking [Nightingale *et al.*, 2011; 2013]. A support of either type is valid iff all literals in it are valid.

A constraint c is Generalised Arc Consistent (GAC) if and only if there exists a full-length support containing every valid literal of every variable in $\text{scope}(c)$. GAC is established by identifying all literals $x \mapsto v$ for which no full-length support exists and removing v from the domain of x. We consider only algorithms for establishing GAC in this paper.

3 Compressing Tables

Arbitrary constraints are typically given to a constraint solver as a list of satisfying tuples. To automatically apply one of the 'short' algorithms SHORTSTR2 or HAGGISGAC, we need a procedure to compress the full-length tuples into short supports. This is equivalent to minimising the size of a DNF formula. This is an NP-complete problem with no polynomial-time approximation scheme [Khot and Saket, 2008]. For practical reasons we designed a greedy approximation algorithm rather than doing optimal compression.

In this section we use full-length tuples to represent short supports, with a special symbol $*$ indicating when a variable is not contained in the short support. Suppose we had a constraint c with scope $\{x_1, x_2, x_3\}$, and a short support $S = \{x_1 \mapsto 1, x_2 \mapsto 2\}$. S is represented as the full-length tuple $\langle 1, 2, * \rangle$. The $*$ indicates that x_3 is not contained in S.

The basic step is to find a set X of tuples where all tuples are equal except at one position i, and at position i we have one tuple for each value in the corresponding domain $D(x_i)$. Thus x_i can take any value and the constraint will still be satisfied, therefore we can create a new short support that does not mention x_i. X can be replaced by one tuple with $*$ at position i, and identical to those in X at all other positions.

Algorithm 1 Greedy-Compress(InTuples)

```
Require: InTuples: Set of full-length tuples.
 1: OutTuples ← ∅
 2: while |InTuples| > 0 do
 3:     UsedTuples ← ∅
 4:     CompTuples ← ∅
 5:     for all $\tau \in$ InTuples do
 6:         if $\tau \notin$ UsedTuples then
 7:             for all $x_i \in scope(c)$ do
 8:                 if $\tau[x_i] = *$ then
 9:                     Continue to next variable (loop on line 7)
10:                 $\sigma \leftarrow \tau$
11:                 found ← true
12:                 for all $v \in D(x_i)$ do
13:                     $\sigma[x_i] \leftarrow v$
14:                     if $\sigma \notin$ InTuples or $\sigma \in$ UsedTuples then
15:                         found ← false
16:                         Break out of loop on line 12
17:                 if found then
18:                     for all $v \in D(x_i)$ do
19:                         $\sigma[x_i] \leftarrow v$
20:                         UsedTuples ← UsedTuples ∪ {$\sigma$}
21:                     $\sigma[x_i] \leftarrow *$
22:                     CompTuples ← CompTuples ∪ {$\sigma$}
23:                     Break out of loop on line 7
24:     OutTuples ← OutTuples ∪ (InTuples \ UsedTuples)
25:     InTuples ← CompTuples
26: return OutTuples
```

Suppose all variables of c are boolean (i.e. $D(x_i) = \{0, 1\}$) and we have three tuples $t_1 = \langle 0, 1, 0 \rangle$, $t_2 = \langle 0, 1, 1 \rangle$ and $t_3 = \langle 1, 1, 0 \rangle$. Tuples t_1 and t_2 may be compressed, and also t_1 and t_3.

The Greedy algorithm starts with full-length supports (i.e. mentioning all r variables) and attempts to compress them to short supports of size $r - 1$ using the basic step described above. It then takes the $r - 1$ short supports and attempts to compress them to size $r - 2$ using the same basic step. This process is repeated until it is not possible to apply the basic step. Pseudocode for Greedy is given in Algorithm 1. The outer while loop iterates for decreasing size, starting with size r. For each size, the algorithm iterates through the tuples of that size (line 5). For each tuple, it iterates through the set of variables that have a value in that tuple (i.e. are not $*$), and for each variable it checks if the tuple can be compressed at that point using other short supports of the same size (lines 10-16). If a set of tuples can be compressed, they are added to UsedTuples (and therefore removed from consideration), and the new (compressed) tuple is added to CompTuples. Finally, Greedy collects up all tuples (of all sizes) that were not consumed by a compression step.

Consider the example above with tuples t_1, t_2 and t_3. Greedy starts with t_1 and considers each position in that tuple starting with position 1. t_1 and t_3 may be compressed to $t_4 = \langle *, 1, 0 \rangle$, because t_1 and t_3 are equal except for position 1, and t_1 and t_3 contain all values in $D(x_1)$. The greedy algorithm removes t_1 and t_3 from consideration, adds t_4, and moves on to t_2. It would be possible to compress t_2 with t_1,

table			Node		
tuple #	$\langle x, y \rangle$		A	B	BT-A
1	$\langle 1, 1 \rangle$	position	1	1	1
2	$\langle 1, 2 \rangle$		2	5	5
3	$\langle 1, 3 \rangle$		3	3	3
4	$\langle 2, 2 \rangle$		4	(4)	4
5	$\langle 2, 3 \rangle$		5	(2)	2
		currentLimit	**5**	**3**	**5**

Table 1: Constraint $x \leq y$ with variable domains $D(x) = \{1, 2\}$ and $D(y) = \{1, 2, 3\}$. table contains all satisfying tuples indexed by $1 \ldots 5$. At node A, all five tuples are in the set. At node B, suppose $y \mapsto 2$ is pruned. Tuple 2 needs to be removed: the values of position[2] and position[5] are swapped, and currentLimit is reduced to 4. Tuple 4 also needs to be removed: there is no need to swap it, and currentLimit is reduced to 3. On backtracking to node A, currentLimit is restored to 5, so tuples 2 and 4 are in the set.

however t_1 has already been used. Next the algorithm moves on to the next size, and tuple t_4. This cannot be compressed so the end result is $\{t_2, t_4\}$.

The complexity of Algorithm 1 is $O(\log_d(t)tr^2d)$. The outer while loop iterates $\log_d(t)$ times, and there are at most d^r tuples therefore $\log_d(t) \leq r$. The loop on line 5 iterates t times. The loop on line 7 iterates r times, and the two loops inside it each iterate d times and contain set operations that are $O(r)$ when using tries for each set. Line 24 is $O(tr)$ and is dominated by the loop above.

4 STR2+

We will describe STR2+ [Lecoutre, 2011] as a propagator embedded in search. As search moves forward, domains are reduced by other propagators and/or the search algorithm, and on backtracking the domains are restored. The key feature of STR2+ is the set of active tuples. Initially this set contains all satisfying tuples. It is reduced incrementally as search moves forward, to remove tuples that are invalid.

table is an array indexed by $1 \ldots t$ containing all satisfying tuples of the constraint.

position is an array indexed by $1 \ldots t$ of integers that index into table.

currentLimit is an integer, the current size of the set. It is initially t.

The table array is static and may be shared between constraints. The tuples in the range position[1..currentLimit] are in the current set, and position[currentLimit+1..t] are outside the set. A tuple position[i] is removed from the current set by swapping position[i] with position[currentLimit], then decrementing currentLimit. On backtracking, only currentLimit is restored, thus the tuple reappears in the current set but the order of position may have changed. This is illustrated in Table 1.

STR2+ does not directly compute domain deletions, but instead constructs a set for each variable of values that are known to be supported. This is done by iterating for all tuples t from position[1] to position[currentLimit]. If t is not valid, it is removed from the set. Otherwise, values in t are added to gacValues because they are supported. Then domains $D(x)$ are updated by removing any values not in gacValues[x].

For efficiency STR2+ employs two other data structures:

Sval The set of variables whose domains have changed since STR2+ was last called.

Ssup Initially the set of variables that have not been assigned by search. As STR2+ progresses, if gacValues[x] $= D(x)$ then x is removed from Ssup.

Sval is used when checking validity of a tuple. Only the literals of variables in Sval are checked, others are guaranteed to be valid because the domain has not changed. Ssup is used in two places. First, when updating gacValues for a tuple t, there is no need to update gacValues[x] if $x \notin$ Ssup. Second, the variables in Ssup are exactly the variables that need to be pruned in the final stage of the algorithm.

Finally, STR2+ has an array named lastSize containing the size of the domain of each variable at the last call. STR2+ updates and backtracks this, whereas STR2 resets the entries to a null value when backtracking, thus STR2+ accurately computes Sval from lastSize and STR2 approximates Sval. The implementation of STR2+ for our experiments does not have lastSize and is given an accurate Sval by the solver.

5 ShortSTR2

The key feature of short supports is that when a short support does *not* contain some variable x, it supports *all* values of x. We call this *implicit* support, as opposed to *explicit* support of literals contained in the short support. Implicit support combined with the fact that there can be many fewer short supports than full-length supports can make our new algorithm SHORTSTR2 much more efficient than STR2+ when the constraint is amenable.

In SHORTSTR2 we use two representations of short support. Suppose we had a constraint with scope $\{x_1, x_2, x_3, x_4\}$, and a short support $S = \{x_1 \mapsto 1, x_2 \mapsto 2\}$. The long form of S is an array $\langle 1, 2, *, * \rangle$ where $*$ indicates that a variable is not mentioned in S. This form allows us to look up the value (or $*$) for a particular variable in $O(1)$ time. The short form is an array of pairs $\langle (x_1, 1), (x_2, 2) \rangle$. This allows us to iterate efficiently through the short support.

SHORTSTR2 shares the table, currentLimit and position data structures with STR2+. In SHORTSTR2, table contains short supports in the long form (with $*$). SHORTSTR2 also shares the Sval and Ssup data structures. It adds one new data structure named shorttable which is an array indexed by $1 \ldots t$ containing the short supports in the short form.

SHORTSTR2 (Algorithm 2) first seeks one valid short support in order to initialise Ssup (lines 2-11). If there are no valid tuples remaining, the constraint is unsatisfiable and the algorithm returns False on line 11.

The main loop of the algorithm is on lines 12-28. It iterates through the position array from 1 to currentLimit, removing any short supports that are no longer valid (lines 15, 27-28). The validity check is performed by Algorithm 3. For those short supports that are valid, the algorithm updates gacValues and Ssup. Ssup is important for efficiency here. The

Algorithm 2 SHORTSTR2-Propagate(Sval)

```
Require: Sval, set of variables with reduced domains
 1: $\mathsf{Ssup} \leftarrow \emptyset$
 2: while $\mathsf{currentLimit} \geq 1$ do
 3:     $posi \leftarrow \mathsf{position}[1]$
 4:     if SHORTSTR2-Valid($posi$, Sval) then
 5:         $\mathsf{Ssup} \leftarrow \{x \mid (x \mapsto a) \in \mathsf{shorttable}[posi]\}$
 6:         Break out of loop on line 2
 7:     else
 8:         Swap position[1] and position[currentLimit]
 9:         $\mathsf{currentLimit} \leftarrow \mathsf{currentLimit} - 1$
10: if $\mathsf{currentLimit} < 1$ then
11:     return False
12: $i \leftarrow 1$
13: while $i \leq \mathsf{currentLimit}$ do
14:     $posi \leftarrow \mathsf{position}[i]$
15:     if SHORTSTR2-Valid($posi$, Sval) then
16:         $\tau \leftarrow \mathsf{table}[posi]$
17:         for all $x_i \in \mathsf{Ssup}$ do
18:             if $\tau[x_i] = *$ then
19:                 $\mathsf{Ssup} \leftarrow \mathsf{Ssup} \setminus \{x_i\}$
20:             else
21:                 if $\tau[x_i] \notin \mathsf{gacValues}[x_i]$ then
22:                     $\mathsf{gacValues}[x_i] \leftarrow \mathsf{gacValues}[x_i] \cup \tau[x_i]$
23:                     if $|\mathsf{gacValues}[x_i]| = |D(x_i)|$ then
24:                         $\mathsf{Ssup} \leftarrow \mathsf{Ssup} \setminus \{x_i\}$
25:         $i \leftarrow i + 1$
26:     else
27:         Swap position[$i$] and position[currentLimit]
28:         $\mathsf{currentLimit} \leftarrow \mathsf{currentLimit} - 1$
29: for all $x_i \in \mathsf{Ssup}$ do
30:     for all $v \in D(x_i)$ do
31:         if $v \notin \mathsf{gacValues}[x_i]$ then
32:             $D(x_i) \leftarrow D(x_i) \setminus v$
```

inner loop on line 17 iterates only for those variables in Ssup. A variable is removed from Ssup whenever all its values are supported, either implicitly (line 19) or explicitly (line 24).

Finally the algorithm prunes the domains of those variables in Ssup, removing any values not in gacValues.

5.1 Complexity of SHORTSTR2

STR2+ and SHORTSTR2 follow a similar pattern. Firstly, the validity of each tuple must be checked, then the supported values of the variables are gathered from the valid tuples, and finally the domains of the variables must be pruned. There are some minor differences. STR2+ makes use of Past(P), the list of variables that have been assigned by the search procedure. In some solvers it is not possible for a propagator to obtain Past(P), so this step is not included in SHORTSTR2. However, any variable x that is assigned has $|D(x)| = 1$, so it will be filtered out of Ssup the first time a valid tuple is found. Second, STR2+ searches through the variables that are not in Past(P) to find those whose domain has changed since the last call. STR2+ also updates and backtracks the lastSize array (containing the domain size of variables on last call). In SHORTSTR2 we assume we can obtain the list of changed variables directly from the solver, therefore we do not need lastSize.

The original complexity analysis of STR2+ contains a small error. Given the set of unassigned (by search) variables

Algorithm 3 SHORTSTR2-Valid($posi$, Sval)

```
Require: $posi$, index into table
Require: Sval, set of variables
 1: $\tau \leftarrow \mathsf{table}[posi]$
 2: for all $x_i \in \mathsf{Sval}$ do
 3:     if $\tau[x_i] \neq * \wedge \tau[x_i] \notin D(x_i)$ then
 4:         return False
 5: return True
```

U, the domain size d and the number of tuples t, the complexity is given as $O(|U| \times d + |\mathsf{Sval}| \times t)$ [Lecoutre, 2011]. This misses the cost in lines $16 - 20$ of STR2+ of $O(|\mathsf{Ssup}|)$ for constructing gacValues. Initially $|\mathsf{Ssup}| = |U|$, giving a total complexity of $O(|U| \times d + (|\mathsf{Sval}| + |U|) \times t)$.

As we described above, SHORTSTR2 does not have access to Past(P). Variables assigned by search can be inserted into Ssup, and they are removed immediately after entering the main loop (lines 12-28) at a cost of r. Therefore we have $O(rd + (|\mathsf{Sval}| + |U|) \times t)$ for SHORTSTR2.

We can improve on this bound by introducing a, the maximum length of a short support. $|\mathsf{Ssup}| \leq a$ because Ssup is created on Line 5 from a short support. This gives us the tighter bound of $O(rd + (|\mathsf{Sval}| + min(|U|, a)) \times t)$.

6 Experimental Evaluation

The Minion solver 0.14 [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 for case studies 1-4, and a 32-core machine with AMD Opteron 6272 CPUs at 2.1GHz and 256GB memory for case study 5. In each of our experiments, all implementations produce an identical search tree, which means we can compare them purely based on the node rate. We search for all solutions for a maximum of ten minutes. Each experiment is run five times, and we give the mean average nodes per second searched. We have not attempted to separate out 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.

The effectiveness of short supports has already been shown [Nightingale *et al.*, 2011; 2013]. Therefore in this paper we concentrate on two main contributions. We compare SHORTSTR2 to HAGGISGAC, and find that SHORTSTR2 is competitive while being much easier to understand and implement. Secondly, we compare STR2+ and SHORTSTR2 on standard benchmarks using tuple compression.

Case studies 1-4 compare SHORTSTR2 and HAGGISGAC on intensional constraints. In some cases there are special purpose propagators that would almost certainly outperform SHORTSTR2. Gent et al. [2010] automatically generated a propagator for the Life constraint that is substantially faster than SHORTSTR2, and for the Vector Not-Equal constraint it is likely that Watched Or [Jefferson *et al.*, 2010] is more efficient than SHORTSTR2. These experiments are to compare SHORTSTR2 and HAGGISGAC, and we save space by omitting other propagators.

n	WatchElt	SHORTSTR2		HAGGISGAC	
	1D	2D	1D	2D	1D
7	7,519	4,433	2,652	5,058	2,971
8	6,347	2,309	1,449	2,927	2,079
9	4,918	1,849	1,142	2,199	1,504
10	4,110	1,324	894	1,703	1,415

Table 2: Nodes per second for QG3. The first column is the Watched Element propagator (WatchElt).

Class	SHORTSTR2		HAGGISGAC	
	Greedy	Full-len	Greedy	Full-len
Life	4,970	3,960	3,030	2,280
Brian's Brain	532	75.0	750	93.0
Immigration	4,930	3,590	1,460	361
Quadlife	483	-	59.7	-

Table 3: Nodes per second for solving the 6×6, 5 step, oscillator problem in four variants of Conway's game of life. A "-" denotes a memory usage of > 4GB

Case Study 1: Element

We use the quasigroup existence problem QG3 exactly as described by Nightingale et al. [2013]. The problem requires indexing a 2D matrix with two variables. We model the indexing in two ways. *1D* is a model with an auxiliary index variable y and a conventional element(X, y, z) constraint. The short supports are 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. The second model, named *2D*, directly represents 2-dimensional indexing using short supports. The set of short supports are of the form $\langle x_{a,b} \mapsto j, y_1 \mapsto a, y_2 \mapsto b, z \mapsto j\rangle$, where a and b index into the $n \times n$ matrix X, and j is a common value of z and $x_{a,b}$. Our results are presented in Table 2. SHORTSTR2 is slower than HAGGISGAC, but is quite close. The 2D model is faster than 1D, for both algorithms. Both SHORTSTR2 and HAGGISGAC are slower than the Watched Element propagator in Minion [Gent *et al.*, 2006b].

The node rates for HAGGISGAC and *WatchElt* on model 1D differ from those reported previously [Nightingale *et al.*, 2013], as we now enforce GAC on the constraints linking the auxiliary index variable to the two indexing variables. GAC ensures an identical search to the *2D* model.

Case Study 2: Oscillating Life

Conway's Game of Life was invented by John Horton Conway. We consider the problem of maximum density oscillators (repeating patterns). We consider Life and three variants of it. Immigration has two *alive* states. 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. Quadlife has four *alive* states. When a cell becomes alive, it takes the state of the majority of the 3 neighbouring live cells which caused it to become alive, unless all 3 neighbours have different colours in which case it takes the colour which none of its neighbours have. Apart from this the rules are the same as Life. Finally Brian's Brain has three states: *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 becomes *dying* after one time step. If a cell is *dying*, it becomes *dead* after one time step.

We use the problem and constraint model as described by Gent et al. [2010]. For all four problems, we make one change: we minimise the occurrences of the value 0 in all layers. Furthermore, for Immigration, Quadlife and Brian's Brain we add extra domain values for each additional state.

In each experiment we use one instance, a 6×6 grid, with 5 time steps. We applied Greedy-Compress to generate the short support sets. Table 3 shows SHORTSTR2 is fastest for Life, Immigration and Quadlife, while HAGGISGAC is fastest on Brian's Brain. SHORTSTR2 is always competitive with and can greatly outperform HAGGISGAC. Short supports are much better than full-length supports with both SHORTSTR2 and HAGGISGAC for all four instances.

$n-w-h$	SHORTSTR2	HAGGISGAC
18-31-69	1,740	7,272
19-47-53	2,114	4,025
20-34-85	1,011	4,416
21-38-88	797	4,917
22-39-98	675	3,636
23-64-68	856	1,889
24-56-88	631	2,421

Table 4: Nodes per second for rectangle packing.

p	a	SHORTSTR2	HAGGISGAC
30	5	92,500	44,100
30	10	142,000	70,700
30	20	111,000	67,000
30	50	87,200	55,000
30	100	67,600	45,200
30	200	53,700	46,100
5	100	592,000	1,790,000
10	100	250,000	653,600
20	100	119,000	158,800
30	100	67,600	45,200
40	100	43,700	18.000
50	100	31,900	10,900

Table 5: Nodes per second for Vector Not-Equal experiment. First section fixes p and increases a. Second section fixes a and increases p.

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 $width \times height$. We use the model (and short support set) described by Nightingale et al. [2013]. Results are shown in Table 4. We can see that in this case HAGGISGAC outperforms SHORTSTR2 by around five times.

Case Study 4: Vector Not-Equal

Disequality between two arrays of decision variables is a useful constraint. For two arrays X and Y of length n, containing variables with domain $\{1 \ldots d\}$, we can express this constraint with the set of short supports $\{\langle X[i] \mapsto j, Y[i] \mapsto k\rangle\}$,

Problem Class	Full Length Supports	Compress Ratio	Greedy+HAGGISGAC setup	Greedy+HAGGISGAC n/s	Greedy+SHORTSTR2 setup	Greedy+SHORTSTR2 n/s	STR2+ setup	STR2+ n/s
half	2,222,907	1.87	158.07	109.87	156.91	639.39	126.87	365.42
modifiedRenault	201,251	5.35	12.15	9,574.9	12.10	13,846.	10.92	14,055.
rand-8-20-5	1,406,195	1.01	115.88	40.29	112.63	3,376.6	90.42	3,207.4
wordsHerald	21,070	1.00	1.37	2,020.4	1.48	2,210.1	1.33	2,216.3
bddSmall	57,756	1.90	12.78	46.44	13.67	587.91	15.41	521.10
renault	194,524	6.31	11.93	211,491.	11.96	591,062.	10.88	555,206.
wordsVg	2,859	1.00	0.56	318.04	0.58	396.06	0.53	396.25
bddLarge	6,895	1.80	5.18	34.74	9.04	26.89	15.84	22.30
cril	7,350	1.19	0.37	2,832.2	0.34	3,080.9	0.31	2,777.4

Table 6: Comparison of Greedy-Compress+SHORTSTR2, Greedy-Compress+HAGGISGAC and STR2+ on classes of CSPXML instances. Setup Time refers to the amount of time taken to load the problem, run Greedy-Compress and initialise each constraint. Compress Ratio refers to the compression ratio of Greedy-Compress, and n/s is the nodes per second achieved during search. Each value reported here is the geometric mean over the instances in the problem class.

for $i \in \{1 \dots n\}, \{j, k\} \subseteq \{1 \dots d\}$. We consider a generalisation of the pigeon-hole problem, where we have p arrays of length a of boolean variables. Each pair of arrays is distinct.

We consider two different experiments. The first, in Table 5, fixes p while increasing a. In this experiment we see that SHORTSTR2 always outperforms HAGGISGAC by a factor of at most two. The second experiment, in Table 5, fixes a while increasing p. For low p, HAGGISGAC is more efficient but as the number of constraints increases SHORTSTR2 becomes more efficient. While neither HAGGISGAC or SHORTSTR2 are even an order of magnitude faster than the other, this experiment shows the potential benefit of a simple algorithm with only a small amount of backtracking state to maintain.

Case Study 5: CSP Solver Competition

The STR2+ paper [Lecoutre, 2011] used a selection of benchmarks from CSP Solver Competitions. These benchmarks can be found at http://www.cril.fr/~lecoutre/. In this experiment we compare SHORTSTR2 and Greedy-Compress to STR2+. We selected classes of problems which have extensional constraints of arity 4 or above and at least 100 tuples per constraint. For each instance of each problem class, we searched for all solutions with a 600 second time limit. As each technique produces exactly the same search tree, we report the setup time and search rate (in nodes per second). The results in Table 6 show that Greedy+SHORTSTR2 compares favourably to STR2+, both in terms of setup time and node rate. The modifiedRenault class is the only exception, it has a good compression ratio but the node rate for SHORTSTR2 is slightly lower in this case. SHORTSTR2 does not always outperform HAGGISGAC, but SHORTSTR2 scales better to large sets of short supports.

The instances that compress poorly are in two categories. The rand-8-20-5 class feature randomly generated constraints that compress only slightly or not at all. The wordsVg and wordsHerald classes contain constraints that represent words in a natural language. This is a natural constraint to represent as a list of tuples, but offers no compression.

This experiment shows that Greedy+SHORTSTR2 provides a valuable tool to users, and it can replace STR2+ without requiring users to understand or provide short supports.

7 Related Work

Simple Tabular Reduction (STR) was first introduced by Ullmann [2007]. It was made more efficient by Lecoutre [2011] who developed new algorithms STR2 and STR2+.

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 full-length valid tuple contained in the c-tuple satisfies the constraint. They outline a modified version of GAC-Schema that directly stores c-tuples, and discuss compression approaches. While c-tuples offer space savings, Katsirelos and Walsh found using c-tuples provides almost no performance gain over GAC-Schema. Régin proposed an even more expressive scheme called tuple sequences [2011]. Both c-tuples and tuple sequences will allow greater compression than short supports, and present an interesting avenue for future work: whether an efficient STR2+-like algorithm can be constructed with c-tuples or tuple sequences.

8 Conclusions

We have introduced SHORTSTR2, a new general-purpose propagation algorithm that makes use of short supports. We also presented a simple greedy algorithm Greedy-Compress to convert a set of satisfying tuples to a set of short supports.

How does SHORTSTR2 *compare to* STR2+*?* In our experiments, we found that SHORTSTR2 combined with Greedy-Compress would make an effective replacement for STR2+, in some cases yielding substantial efficiency gains.

How does SHORTSTR2 *compare to* HAGGISGAC*?* We chose HAGGISGAC as the comparison because it is clearly superior to SHORTGAC, and HAGGISGAC is simpler than HAGGISGAC-STABLE while having very similar performance. The comparison between SHORTSTR2 and HAGGISGAC is mixed, with each algorithm having its own strengths and weaknesses. They are complementary, and SHORTSTR2 is much simpler to implement.

Acknowledgements

We would like to thank EPSRC for funding this work through grant EP/H004092/1, reviewers for their helpful comments, and our colleagues Ian Gent and Ian Miguel for helpful discussions.

References

[Bessière and Régin, 1997] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings IJCAI 1997*, pages 398–404, 1997.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Cheng and Yap, 2010] Kenil C. K. Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[Gent *et al.*, 2006a] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings ECAI 2006*, pages 98–102, 2006.

[Gent *et al.*, 2006b] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In *Proceedings CP 2006*, pages 182–197, 2006.

[Gent *et al.*, 2010] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Generating special-purpose stateless propagators for arbitrary constraints. In *Principles and Practice of Constraint Programming (CP 2010)*, pages 206–220, 2010.

[Jefferson *et al.*, 2010] Christopher Jefferson, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Implementing logical connectives in constraint programming. *Artificial Intelligence*, 174(16-17):1407–1429, 2010.

[Katsirelos and Walsh, 2007] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings CP 2007*, pages 379–393, 2007.

[Khot and Saket, 2008] Subhash Khot and Rishi Saket. Hardness of minimizing and learning DNF expressions. In *In Proc. 49 th IEEE FOCS*, pages 231–240, 2008.

[Lecoutre, 2011] Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[Nightingale *et al.*, 2011] Peter Nightingale, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Exploiting short supports for generalised arc consistency for arbitrary constraints. In *Proceedings IJCAI 2011*, pages 623–628, 2011.

[Nightingale *et al.*, 2013] Peter Nightingale, Ian P. Gent, Christopher Jefferson, and Ian Miguel. Short and long supports for constraint propagation. *Journal of Artificial Intelligence Research*, 46:1–45, 2013.

[Régin, 2011] J.C. Régin. Improving the expressiveness of table constraints. In *The 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*, 2011.

[Simonis and O'Sullivan, 2008] Helmut Simonis and Barry O'Sullivan. Search strategies for rectangle packing. In *Proceedings CP 2008*, pages 52–66, 2008.

[Ullmann, 2007] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177:3639–3678, 2007.