A compression algorithm for large arity extensional constraints*

George Katsirelos and Toby Walsh

NICTA and UNSW
[george.katsirelos, toby.walsh]@nicta.com.au

Abstract. We present an algorithm for compressing table constraints representing allowed or disallowed tuples. This type of constraint is used for example in configuration problems, where the satisfying tuples are read from a database. The arity of these constraints may be large. A generic GAC algorithm for such a constraint requires time exponential in the arity of the constraint to maintain GAC, but Bessière and Régin showed in [1] that for the case of allowed tuples, GAC can be enforced in time proportional to the number of allowed tuples, using the algorithm GAC-Schema.
We introduce a more compact representation for a set of tuples, which allows a potentially exponential reduction in the space needed to represent the satisfying tuples and exponential reduction in the time needed to enforce GAC. We show that this representation can be constructed from a decision tree that represents the original tuples and demonstrate that it does in practice produce a significantly shorter description of the constraint. We also show that this representation can be efficiently used in existing algorithms and can be used to improve GAC-Schema further. Finally, we show that this method can be used to improve the complexity of enforcing GAC on a table constraint defined in terms of forbidden tuples.

1 Introduction

The table constraint is an important constraint that is available in most constraint toolkits. With this, we are able to express directly a set of acceptable assignments to a set of variables. These constraints can be generated from data that has been read from a database in a configuration problem, or may encode users' preferences, among other applications. The table constraint is usually propagated using GAC-Schema, an algorithm proposed in [1] and studied further in [9, 8, 4] among others.

In this paper, we introduce a compression algorithm for table constraints. This algorithm attempts to capture the structure that may exist in a table but

* NICTA is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program. Thanks to Fahiem Bacchus and Nina Narodytska for their insightful comments.

is not explicitly encoded. In order to achieve this, we propose an alternative representation of tuples that may capture an exponential number of tuples in polynomial space, although each compressed tuple may be larger than the arity n of the constraint. GAC-Schema can be adapted easily to work with such compressed tuples. Since the runtime of GAC-Schema is proportional to the number of tuples in the table and we do not increase the cost of examining a single tuple significantly, we can reasonably expect it to perform better after we reduce the number of tuples in this way. The approach can also be extended from the sets of allowed tuples to work also on constraints that are expressed as sets of forbidden tuples. This can potentially deliver great improvements in runtime, as the complexity of GAC-Schema+Forbidden is exponential in the arity of the constraint, whereas with our approach it becomes proportional to the number of forbidden tuples.

The rest of the paper is organized as follows. We first present the necessary background in section 2, then we describe the compression algorithm in section 3. In section 4 we describe how GAC-Schema can be modified to work on compressed tables. In section 5, we extend this approach to work on tables of forbidden tuples. Finally, in section 6 we present experimental confirmation that GAC-Schema using compressed tuples is faster.

2 Background

A constraint satisfaction problem $\mathcal{P}$ is a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V}$ is a set of variables, $\mathcal{D}$ is a function mapping a variable to a domain of values and $\mathcal{C}$ is a set of constraints. Given $V \in \mathcal{V}$ and $x \in \mathcal{D}(V)$, we say that $V = x$ is an *assignment*, it assigns to V the value x. The goal is to find a set of assignments that *satisfy* the constraints, assigning exactly one value to each variable.

An **assignment set** is a set of assignments $\mathcal{A} = \{X_1 = a_1, \ldots, X_k = a_k\}$ such that no variable is assigned more than one value. We use $scope(\mathcal{A})$ to denote the set of variables assigned values in $\mathcal{A}$. A constraint C consists of an ordered set of variables, $scope(C)$, and a set of assignment sets. Each of these specifies an assignment to the variables of $scope(C)$ that satisfies C. We say that an assignment set $\mathcal{A}$ is **consistent** if it satisfies all constraints it covers: $\forall C. scope(C) \subseteq scope(\mathcal{A}) \Rightarrow \exists \mathcal{A}'. \mathcal{A}' \in C \wedge \mathcal{A}' \subseteq \mathcal{A}$. Thus, a **solution** to the CSP is a consistent assignment set containing all of the variables of the CSP.

We deal here with backtracking search algorithms. Constraint propagation is used during backtracking search to filter the domains of variables so that values that cannot be part of a solution are removed from the domains of unassigned variables. Most solvers maintain generalized arc consistency for the table constraint.

Definition 1 (Support). *A* support *of a constraint C is a set of assignments to exactly the variables in $scope(C)$ such that C is satisfied. A support of C that includes the assignment $V = x$ is called a support of $V = x$ in C.*

Definition 2 (Generalized arc consistency (GAC)). *A constraint C is GAC if there exists a support for all values in the current domains of the variables in scope(C). A problem P is GAC if all of its constraints are GAC.*

Constraints can be defined in several ways, including as a set of satisfying assignments; as a set of non-satisfying assignments; as a predicate; or algebraically. In this paper, we deal with the first two (extensional) representations.

3 The compression algorithm

Representation. Let C be a constraint that is represented in extension as a set of satisfying tuples. Let $\mathcal{U}_C = \{U_1, \ldots, U_u\}$ be the set of satisfying tuples that define the constraint, $n = |scope(C)|$ and $d = max_{V \in scope(C)} |\mathcal{D}(V)|$.

The set of tuples $\mathcal{U}_C$ represents the propositional disjunction $c(U_1) \vee \ldots \vee c(U_u)$, where $c(U_i)$ is $V_1 = d_{i1} \wedge \ldots \wedge V_n = d_{in}$, the propositional form of the tuple $U_i = \langle d_{i1}, \ldots, d_{in} \rangle$. From now on, we refer to these as u-tuples. The compression scheme that we propose transforms the constraint into a conjunction of compressed tuples. Each compressed tuple C_i corresponds to a more compact propositional formula $c(C_i)$ of the form $(V_1 = d_{i,1,1} \vee \ldots \vee V_1 = d_{i,1,k_{i,1}}) \wedge \ldots \wedge (V_n = d_{i,n,1} \vee \ldots \vee V_n = d_{i,n,k_{i,n}})$. A compressed tuple can then be written $C_i = \langle (d_{i,1,1}, \ldots, d_{i,1,k_{i,1}}), \ldots, (d_{i,n,1}, \ldots, d_{i,n,k_{i,n}}) \rangle$. A compressed tuple C_i admits any set of assignments that assigns one of $d_{i,1,1}, \ldots, d_{i,1,k_{i,1}}$ to V_1, one of $d_{i,2,1}, \ldots, d_{i,2,k_{i,1}}$ to V_2 and so on. Note that a compressed tuple can represent a potentially exponential number of u-tuples. Specifically it will accept any combination of the $k_{i,1}$ values for V_1, $k_{i,2}$ values for V_2 and so on, so that it represents $k_{i,1} k_{i,2} \ldots k_{i,n}$ u-tuples. We refer to this representation as c-tuples. The set of u-tuples represented by a c-tuple c is written $u(c)$.

This compact representation is not new, as it has been used before in different contexts. Milano and Focacci[3] used it in the context of symmetry breaking, while Katsirelos and Bacchus[6] used it in the context of nogood learning.

Clearly not all sets of tuples can be compressed in this way. For any set of domains, there exist two maximum sets of tuples that cannot be compressed. We derive them by ordering all possible tuples lexicographically and choosing those with an even (odd) index in this ordering. In general, a set of tuples can be expressed more succinctly as c-tuples if there exist clusters of tuples with pairwise Hamming distance 1. For example, consider the set $U_1 = \{\langle 1,1,1 \rangle, \langle 1,2,2 \rangle\}$. Since the two tuples have Hamming distance 2, this cannot be expressed any more succinctly as a set of c-tuples. The set $U_2 = \{\langle 1,1,1 \rangle, \langle 1,2,1 \rangle, \langle 1,2,2 \rangle\}$ has two pairs of tuples with Hamming distance 1, while the other two have distance two, and can therefore be expressed either as $C_2 = \{\langle (1)(1,2)(1) \rangle, \langle (1)(2)(2) \rangle\}$ or $C_2' = \{\langle (1)(1)(1) \rangle, \langle (1)(1)(1,2) \rangle\}$. Finally, by adding one more tuple, the set $U_3 = \{\langle 1,1,1 \rangle, \langle 1,1,2 \rangle, \langle 1,2,1 \rangle, \langle 1,2,2 \rangle\}$, all tuples have pairwise Hamming distance 1 and can be expressed as the single c-tuple $\langle (1)(12)(12) \rangle$.

Note this representation cannot improve the efficiency of existing algorithms (e.g. AC-*[12]) for propagating binary constraints. Since the arity of these constraints is two, each compressed tuple can at best represent a quadratic number

of uncompressed tuples. However, when AC-* examines a value to determine whether it still has a support, it will only examine the tuples that contain this value. There is a linear number of such tuples, which can each be checked in constant time. On the other hand checking the single compressed tuple is done in linear time also, so in the best case using compressed tuples for binary constraints yields no improvement.

The applicability of the representation is also reduced for tables where some of the variables are functionally dependent on some others. Consider for example the arithmetic constraint $X = Y + Z$. For every value of Y and Z, X is uniquely defined. Thus, we can only attempt to represent more compactly the set of values of Y and Z that yield a specific value of X. The functional dependency of X on Y and Z effectively partitions the space of possibly more compact representations among the different values of X and reduces the arity of the table by 1, as far as the compression scheme is concerned.

Decision trees. We derive a set of c-tuples from the original u-tuples by creating a decision tree $T(U)$ that describes the u-tuples and then deriving c-tuples from the decision tree.

A decision tree is a structure that can describe a Boolean function. Each non-leaf node v of the tree is labeled with a test $l(v)$. Each leaf is labeled with either 1 or 0. In order to evaluate the Boolean function $f(\boldsymbol{x})$, we start at the root and test x against $l(v)$. The left child of v is visited if the test succeeds and right child if it fails. When a leaf is reached, the value of $f(\boldsymbol{x})$ is determined to be the same as the label of the leaf. For convenience, we also label the edge to the left child of v with $l(v)$ and the edge to the right child of v with $\neg l(v)$.

Here we treat a set of tuples $\mathcal{U}_C$ as a function $f_{\mathcal{U}_C} : \{0,1\}^{dn} \to \{0,1\}$ where the tests are literals that correspond to all dn possible assignments. f evaluates to 1 if its arguments form a tuple in $\mathcal{U}_C$ and 0 otherwise. Let $T(\mathcal{U}_C)$ be a decision tree that describes $f_{\mathcal{U}_C}$. A u-tuple $u \in \mathcal{U}_C$ *agrees* with a node v if v is the root or if it agrees with the parent $P(v)$ of v and v is the left child of $P(v)$ and $l(P(v)) \in u$ or v is the right child of $P(v)$ and $l(P(v)) \notin u$. A tuple $u \in \mathcal{U}_C$ is associated with a node v if it agrees with it and the set of all such tuples is $U(v)$. By this definition, each u-tuple agrees with at most one of the child nodes of v. Additionally, we enforce that each tuple $u \in U(v)$ must agree with exactly one child of v[1] so all the nodes at a level of the tree describe a partition of $\mathcal{U}_C$. If $U(v) = \emptyset$ then the node is *empty*. A literal s (resp. $\neg s$) is *implied* in v iff $\forall u \in U(v)\ s \in u$ (resp. $\forall u \in U(v)\ s \notin u$.)

We can determine the set of tuples $poss(v)$ *described* by a node v by enumerating all the possible u-tuples that agree with v. A node v *completely describes* the set of tuples $U(v)$ associated with it if $poss(v) = U(v)$.

Finally, a c-tuple can be constructed from a node v. Let $S(v)$ be the set of literals that label the edges in the path from v to the root of the tree. $S(v, V)$ is the set of values of V that appear in a literal in $S(v)$ if $\nexists d | V = d \in S(v)$, else it is $\mathcal{D}(V) - \{d\} \,|V = d \in S(v)$. Then the c-tuple $c(v)$ that describes exactly the same

[1] If this is not true, then $T(\mathcal{U}_C)$ is not a decision tree for the function $f_{\mathcal{U}_C}$.

tuples as v is $c(v) = \langle (d_1 \in \mathcal{D}(V_1) - S(v, V_1)) \dots (d_n \in \mathcal{D}(V_n) - S(v, V_n)) \rangle$. For example, if $S(v) = \{V_1 = 1, V_2 \neq 2, V_2 \neq 3, V_3 \neq 1, V_3 \neq 2, V_4 = 2\}$ and $\mathcal{D}(V_1) = \mathcal{D}(V_2) = \mathcal{D}(V_3) = \mathcal{D}(V_4) = \{1, 2, 3, 4\}$, then $c(v) = \langle (1)(1, 4)(3, 4)(2) \rangle$. We see that $c(v)$ admits all u-tuples in $poss(v)$.

Proposition 1. *Let v be a node in the decision tree. The c-tuple $c(v)$ admits exactly the tuples $poss(v)$.*

Proof. Let u be a tuple in $poss(v)$. We only need to show that for each variable V, u assigns d to V if and only if $V = d$ is part of $c(v)$. Since $u(c(v))$ contains all combinations of assignments to each variable, this is enough to show $u \in u(c(v)) \iff u \in poss(v)$ for all u. Let u assign the value d to variable V. Since u agrees with v then either $V = d \in S(v)$ or $V \neq d \notin S(v)$. In the first case, $c(v)$ also assigns d to V. In the second case, $c(v)$ will contain all assignments to V in $D(V) - S(v, V)$. But since $V \neq d \notin S(v)$ then $d \in D(V) - S(v, V)$. Therefore $V = d \in c(v)$. □

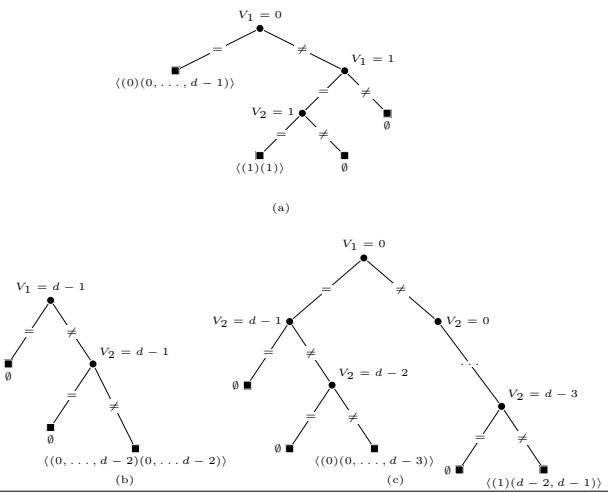
If v completely describes $U(v)$, $c(v)$ also completely describes $U(v)$ and the u-tuples in $U(v)$ can be replaced by $c(v)$. This means that given a decision tree that represents a set of clauses, the set of c-tuples that are constructed from the non-empty leaves of the tree are exactly equivalent to the original u-tuples.

Example Let $C_1(V_1, V_2)$ be a constraint such that $D(V_1) = D(V_2) = \{0, \dots, d-1\}$ and $U(C_1) = \{\langle 0, 1\rangle, \dots, \langle 0, d-1\rangle, \langle 1, 1\rangle\}$. This constraint allows every tuple where $V_1 = 0$ and the tuple $\langle 1, 1\rangle$. In figure 1(a) we show an optimal decision tree and the tuples associated with each leaf node. We also show in 1(b) and 1(c) the decision trees for $C_2(V_1, V_2)$ with $U(C_2) = \{\langle 0, 0\rangle, \dots, \langle 0, d-2\rangle, \langle 1, 0\rangle, \dots, \langle 1, d-2\rangle, \dots, \langle d-2, 0\rangle, \dots, \langle d-2, d-2\rangle\}$ and $C_3(V_1, V_2)$ with $U(C_3) = \{\langle 0, 0\rangle, \dots, \langle 0, d-3\rangle, \langle 1, d-2\rangle, \langle 1, d-1\rangle\}$, respectively. In this figure, a round node v is an internal node of the decision tree and is labeled with the literal that we branch on at v. A square node v is a leaf node and is either labeled by $\emptyset$, in which case is empty, or it completely describes the tuples $U(v)$ and it is labeled with the corresponding c-tuple. The left child of a node that branches on s contains tuples that contain s and thus the edge to the parent is labeled with $=$, while the right child contains tuples that do not contain s and the edge to the parent is labeled with $\neq$.

Constructing decision trees. The problem of constructing a minimum decision tree (with minimum average branch length) is NP-hard [5]. We have tried to solve this problem to optimality using constraint programming techniques, but were unsuccessful in improving significantly over generate-and-test. Thus, we use a heuristic approach.

Algorithms that construct decision trees follow an outline similar to that of TableToDecisionTree, shown in figure 2 (see, for example [10]). This algorithm is straightforward. At each node it checks whether any implied literals exist and extends the tree with a node for each of them. If no implied literals exist, it selects a literal to branch on and then expands each of the child nodes. If

Fig. 1 Optimal decision trees for the constraints (a) C_1, (b) C_2 and (c) C_3. Each node is labeled with the literal on which to branch. Round nodes contain both tuples and non-tuples, while square nodes are leaves and are either empty and labeled with $\emptyset$ or contain tuples only and are labeled with a c-tuple.



it creates an empty node (where $|U(v)| = 0$) or a node that completely describes $U(v)$, it stops. After construction of the tree, a c-tuple is generated from each leaf v that completely describes $U(v)$.

The complexity of the algorithm depends on how much work the splitting heuristic needs to do to select a literal to branch on. Here, we only deal with heuristics that examine the frequency with which literals appear in $U(v)$. If V_l is the set of nodes at level l of the binary tree, then we know that the sets $U(v), v \in V_l$ are a partition of $\mathcal{U}_C$, therefore at each level of the tree each tuple in $\mathcal{U}_C$ will be examined exactly once. Since all tuples have length n, the cost of choosing a literal to branch on for every node at a level is $O(|\mathcal{U}_C|n)$. The depth of the tree is bounded by the total number of literals nd, since a literal cannot be branched on more than once in the same branch. Thus the total cost of building the tree is $O(|\mathcal{U}_C|n^2d)$. Note however that this assumes that no compression can be achieved on the table. In practice the runtimes are often much lower.

Splitting heuristics. The splitting heuristics we will examine are based on counting the frequency with which literals appear in $U(v)$. We write $f(s, v)$ for the number of times s appears in a tuple in $U(v)$ and $f(\neg s, v) = |U(v)| - f(s, v)$. We describe the following heuristics.

Fig. 2 Compression algorithm

```
TableToDecisionTree(U_C, v:Node)
 1.  if v is empty ∨ v is complete
 2.    return
 3.  if ∃ s s.t. s is implied
 4.    v' = {Parent:v, EdgeLiteral:s}
 5.    TableToDecisionTree(U(v'), v')
 6.  else
 7.    s =ChooseLiteral(U(v))
 8.    v1 = {Parent:v, EdgeLiteral:s}
 9.    v2 = {Parent:v, EdgeLiteral:¬s}
10.    TableToDecisionTree(U(v1), v1)
11.    TableToDecisionTree(U(v2), v2)
```

- MaxFreq. This heuristic chooses the literal with maximum $f(s, v)$ (or minimum $f(\neg s, v)$). The reasoning is that the c-tuples that will be constructed in the subtree containing s have a better chance of representing more u-tuples, since s appears often. For example, consider the constraint C_1 discussed earlier, with $U(C_1) = \{\langle 0, 0\rangle, \ldots, \langle 0, d\rangle, \langle 1, 1\rangle\}$, with an optimal decision tree shown in figure 1(a). Since $V_1 = 0$ appears d times, MaxFreq will make the optimal choice at the root. The branch that contains $V_1 = 0$ completely describes all u-tuples for which $V_1 = 0$, thus the algorithm stops expanding at this point. The other branch only contains the tuple $\langle 1, 1\rangle$ which will be left uncompressed. Thus this heuristic finds an optimal branching for this example. On the other hand, consider the constraint C_2 with $U(C_2) = \{\langle 0, 0\rangle, \ldots, \langle 0, d-2\rangle, \langle 1, 0\rangle, \ldots, \langle 1, d-2\rangle, \ldots, \langle d-2, 0\rangle, \ldots, \langle d-2, d-2\rangle\}$, whose optimal decision tree is shown in figure 1(b). Each literal in this set of u-tuples appears exactly $d-1$ times, except for $V_1 = d-1$ and $V_2 = d-1$ which do not appear at all, so MaxFreq chooses one of those that appear $d-1$ times arbitrarily. Assume it chooses $V_1 = 0$. On the positive branch, each literal of V_2 appears once except for $V_2 = d-1$, so once again MaxFreq chooses one arbitrarily. This continues until each c-tuple containing $V_1 = 0$ is placed on a separate branch, therefore no compression occurs. On the negative branch of $V_1 = 0$, each literal of V_1 appears $d-1$ times and each literal of V_2 appears $d-2$ times, so another literal of V_1 is chosen and the process repeats so that no compression occurs. However, $U(C_2)$ is compressible to the single c-tuple $\langle (0, \ldots, d-2)(0, \ldots, d-2)\rangle$, so clearly MaxFreq does not perform optimally in every case.
- MinFreq. This chooses the literal with the minimum $f(s, v)$. If many u-tuples are similar but contain many different assignments to a single variable V, then branching on all the values of V that *do not* appear in the u-tuples will quickly lead to a subtree that contains these similar u-tuples that can hopefully be compressed efficiently. It is easy to see that for the constraint C_2 mentioned above MinFreq will find the optimal compression. On the other hand, it will not do so for the constraint C_1.

- MinMinFreq. This chooses the literal with the minimum $min(f(s,v), f(\neg s,v))$. This is an attempt to combine MinFreq and MaxFreq, by using $f(s,v)$ and $f(\neg s,v)$ respectively as *measures* of the fitness of a literal. At each node, the best literals chosen by both heuristics are compared against each other using this measure and the best one is branched on. We can see that both C_1 and C_2 are compressed optimally using MinMinFreq. However, for the constraint C_3 with $U(C_3) = \{\langle 0,0\rangle, \ldots, \langle 0,d-3\rangle, \langle 1,d-2\rangle, \langle 1,d-1\rangle\}$ (decision tree in figure 1(c)), we have $f(V_1 = 0) = d-2, f(V_1 = 1) = 2$ and $f(V_2 = k) = 1$ for all k. In order to produce the optimum compression $\{\langle (0)(0,\ldots,d-3)\rangle, \langle (1)(d-2,d-1)\rangle\}$, the heuristic needs to branch either on the literal $V_0 = 0$ or on $V_0 = 1$. However, MinMinFreq will not choose either of these and neither will MinFreq. MaxFreq will choose the correct literal to branch on, but we showed that it performs worse in C_2.
- MinDiff. This chooses the literal with the minimum $|f(s,v) - f(\neg s,v)|$. This heuristic tries to create a more balanced tree, therefore a smaller one. MinDiff will perform optimally on C_1 and C_3 but not C_2.
- MaxGain. This is the heuristic used by ID3 [10] and C4.5 [11]. It calculates first the information content $I(v)$ for the current node. For each literal s, it calculates the expected information $E(s)$ of the subtree that will be created by branching on s and chooses s so that $I(v) - E(s)$ is maximum. This means that it chooses the literal that gains the most information. The decision tree under the node v is treated as a source of a message 'T' or 'N' for tuples that belong or do not belong, respectively, to $U(v)$. Then the information of that tree is

$$I(v) = -\frac{|U(v)|}{|poss(v)|}\log_2\frac{|U(v)|}{|poss(v)|} - \frac{|poss(v) - U(v)|}{|poss(v)|}\log_2\frac{|poss(v) - U(v)|}{|poss(v)|}$$

  The expected information required if we branch on the literal s is

$$E(s) = \frac{|poss(v \cup \{s\}))|}{|poss(v)|} I(v \cup \{s\}) + \frac{|poss(v \cup \{\neg s\})|}{|poss(v)|} I(v \cup \{\neg s\})$$

  where $n \cup \{s\}$ and $n \cup \{\neg s\}$ are the child nodes of v resulting by branching on s.
  Since $I(v)$ is common to all literals, MaxGain chooses the literal that minimizes $E(s)$.

Empirical results. We implemented and tested this compression algorithm on all the instances from the 2005 CSP competition [13] that contain non-binary tables. We tested for compression efficiency and runtime performance. We omitted binary instances and binary table constraints in non-binary instances, because as we already pointed out, modern algorithms for maintaining arc consistency in binary constraints cannot benefit from compression.

In table 1 we show the compression achieved with three of the heuristics we mentioned (MinMinFreq, MinDiff and MaxGain) for some of the instances

Table 1. Compression efficiency for instances from the 2005 CSP competition. The best ratios t/t_C and l/l_C are highlighted.

				MinMinFreq			MinDiff			MaxGain		
Instance	# Tables	Avg t	Avg l	t/t_c	l/l_c	time	t/t_c	l/l_c	time	t/t_c	l/l_c	time
Golomb-12-sat	4	2631.25	7893.75	**1.0**	**1.0**	1.16	**1.0**	**1.0**	2.24	**1.0**	**1.0**	2.72
Golomb-12-unsat	8	4776.88	14330.62	**1.0**	**1.0**	2.05	**1.0**	**1.0**	3.15	**1.0**	**1.0**	4.36
0-TSP-10	3	7651.0	22953.0	**1.0**	**1.0**	12.45	**1.0**	**1.0**	24.78	**1.0**	**1.0**	26.46
series10	1	90.0	270.0	1.0	1.0	0.0	**1.36**	**1.22**	0.0	**1.36**	**1.22**	0.0
cril_sat_nb_0	27	1482.11	15076.22	**19.65**	**13.14**	0.01	1.98	1.73	0.03	5.25	4.21	0.01
cril_unsat_nb_6	9	281.33	1125.33	**50.6**	**18.06**	0.0	29.03	9.6	0.0	38.36	12.76	0.0
cril_unsat_nb_7	9	292.67	1170.67	**61.61**	**20.04**	0.0	40.28	11.58	0.0	45.56	13.59	0.0
gr_55_11_a3	1.0	1540	4620.0	**1.0**	**1.0**	0.17	**1.0**	**1.0**	0.09	**1.0**	**1.0**	0.23
random-3-20-20-60-632-forced-1	60	2944.0	8832.0	3.14	1.9	0.06	6.85	2.32	0.04	**7.18**	**2.35**	0.06
random-3-24-24-76-632-forced-1	74	5001.28	15003.85	3.6	1.99	0.13	8.15	2.4	0.07	**8.48**	**2.42**	0.11
random-3-28-28-93-632-forced-1	91	7967.01	23901.03	4.13	2.08	0.25	9.56	2.47	0.15	**9.97**	**2.49**	0.21
renault-merged	89	2182.76	14455.07	7.32	4.08	0.04	**51.92**	**7.65**	0.01	14.34	5.53	0.04

we tested. For each instance, we report the number of tables that were compressed, the average number of tuples t and the average number of literals l per table and for each splitting heuristic the ratios t/t_c, l/l_c and the average time to compress a table. The ratio t/t_c is the ratio of the number of tuple in the original table versus the number of tuples in the compressed table, while l/l_c is the ratio of the total number of literals in the expression. Note that for each constraint C, l is simply $t \cdot |scope(C)|$. The ratio t/t_c gives an indication to how efficient the compression was. l/l_c is an accurate representation of the memory savings that have been achieved.

We show only one representative instance from most families. The results tend to be the same for all instances of one family, as they tend to reuse the same tables. The exception to this is the Golomb ruler problem where the tables express arithmetic tables over domains that grow with the number of marks on the ruler, therefore the tables themselves grow as well. We show results for the largest satisfiable and largest unsatisfiable instance of the Golomb ruler problem. The `cril` family of problem also does not follow this pattern, because it is a collection of instances from different domains.

We see that not all domains are amenable to applying our technique. This is to be expected, as it is possible that the tuples cannot be more compactly represented. On the other hand, for three domains the technique works very well and for those we present results for more instances. In the domains where the technique achieves no reduction in the number of tuples, the overhead of attempting and failing to compress the tables is generally small. The only families where it is even noticeable are the Golomb ruler and travelling salesman problems, in which case it takes 20 seconds and 60 seconds, respectively, to examine all tables of the instance. Moreover, in many cases the tables are shared among many instances and therefore compression can be considered an offline procedure that can be performed once before solving a set of instances.

In the three domains where compression achieves improvement, we see that no splitting heuristic dominates. Moreover, the differences can be dramatic. In the `cril` family the heuristic MinMinFreq outperforms both the other ones and in the instance `cril_sat_nb_0` the ratio l/l_c is 7 times higher than for MinDiff and 3 times higher than for MaxGain. On the other hand, in the `renault` con-

figuration problem, MinDiff performs best and in random problems MaxGain is the better choice.

In our experiments, we also tried to use the C4.5 algorithm [11]. Although the ideas for the construction of decision trees are similar, C4.5 was created with machine learning applications in mind, where the purpose is to improve the ability of the decision tree to correctly classify tuples that have not yet been seen. As a result, the software is not suitable for our purposes. We believe however that our use of the MaxGain heuristic captures the major ideas behind C4.5.

4 Modifying GAC-Schema

The GAC-Schema algorithm was proposed in three different variations: GAC-Schema+Allowed, GAC-Schema+Forbidden, GAC-Schema+Predicate. The first two are intended to work with constraints expressed in extension by satisfying or conflicting tuples, respectively. The third works with constraints defined by a predicate which succeeds for satisfying complete assignments. Here, we only deal with GAC-Schema+Allowed (to which we refer simply as GAC-Schema). In section 5, we will also deal with GAC-Schema+Forbidden.

GAC-Schema uses the procedure seekNextSupport to identify a support for a value $V = x$. seekNextSupport iterates over the tuples in the table that contain $V = x$ until it finds a support for $V = x$, i.e. a tuple such that none of its values have been pruned. In order to minimize the number of checks performed by seekNextSupport, GAC-Schema maintains a *current support* for each unpruned value. A current support for $X = a$ is a tuple that contains $X = a$ and is *valid*. In the context of uncompressed tuples, a tuple is valid if none of the values that it contains are pruned. It maintains this information in three data structures: $S(U)$ is the set of values that are currently supported by the tuple U; $S_C(X = a)$ is the set of tuples that contain $X = a$ and are currently supports for some values; $last_C(X = a)$ is the last support of value $X = a$ returned by seekNextSupport. When a value $X = a$ is pruned, new support has to be found for values that have lost their current support because of this pruning. These values are in the set $P = \bigcup_{U \in S_c(X=a)} S(U)$. For each value $Y = b \in P$, a new support is first sought among $S_c(Y = b)$ and then among tuples that contain $Y = b$ and have index higher than $last_C(Y = b)$. If no support is found by either procedure, the value is pruned. If a support U is found in $S_C(Y = b)$, then $Y = b$ is placed in $S(U)$. If a support σ is found by seekNextSupport, it becomes the new current support for $Y = b$ and the data structures are updated accordingly. Multidirectionality is exploited by placing the new support σ in $S_C(Z = c)$ for every other value in σ.

In order to work on a table of compressed tuples, we first modify the definition of a valid tuple, which is needed by seekNextSupport. A c-tuple is invalid when there exists a variable in the scope of the constraint such that all its values in the c-tuple are pruned. Note that this allows for a simple but significant optimization: if all values of a variable appear in a compressed tuple, we can

simply avoid checking them altogether. However, the complexity of checking whether a c-tuple is valid is $O(nd)$, as a c-tuple can contain many values from each variable, as opposed to $O(n)$ for a u-tuple. On the other hand, the greater the length of the c-tuple, the more u-tuples it represents, so we can expect that the greater complexity of performing a single constraint check is balanced by the fact that we need to perform fewer of them.

In order to have GAC-Schema work with c-tuples, we need to note the following. First, when we prune a value $X = a$, it is not necessary that all tuples in $S_C(X = a)$ will be invalid, as they may contain other values of X that have not been pruned. Second, since a c-tuple is valid as long as one value from each variable is not pruned, it is not necessary to examine the tuple for validity every time one of the values it contains is pruned. We see that the structure $S_C(X = a)$ serves two purposes: it identifies tuples that may become invalid if $X = a$ is pruned and it identifies tuples that have already been found to support other values and may be a support for $X = a$. In the case of uncompressed tuples, the two are necessarily identical, as the pruning of a value $X = a$ will invalidate all tuples $X = a$ appears in. In the case of compressed tuples, we need to maintain two different structures for the two purposes. The first one is called W_C and the second S_C. W_C will always be a (not necessarily strict) subset of S_C. Every time a tuple σ is identified as a support for a variable, we choose one value $X = a \in \sigma$ from each variable in the scope of the constraint and place σ in $W_C(X = a)$. This optimization is similar in spirit to the watch literal technique used in SAT solvers to unit propagate propositional clauses.

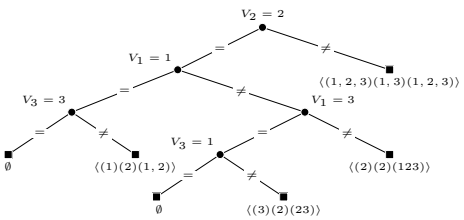
5 GAC-Schema with forbidden tuples

Recall that in constructing a decision tree to compress a set of tuples, we view this set as a Boolean function that evaluates to 1 if its arguments form a tuple in U and 0 otherwise. If we were to use this method on a constraint that is represented as a set of forbidden tuples, we would have no useful way of using the compressed tuples to perform propagation, as the instantiation of the function seekNextSupport for GAC-Schema+Forbidden does not iterate over the tuples, but needs to check whether or not an arbitrary tuple is forbidden.

Note however that we can convert between the two equivalent representations of a constraint as a set of forbidden or allowed tuples. A constraint on variables $V_1, \ldots, V_n$ with domains $D_1, \ldots, D_n$ represented as the set of forbidden tuples U is equivalent to a constraint represented by the set of allowed tuples $D_1 \times \ldots \times D_n - U$. In terms of the corresponding Boolean function, this means that the function evaluates to 1 if its arguments form a tuple not in U and 0 otherwise. This suggests that we can use our compression method to compress the equivalent constraint that is expressed as a set of satisfying tuples, without actually generating the satisfying tuples. This is done by generating c-tuples from the leaves that are empty, as opposed to those that completely describe $U(v)$. These compressed tuples are allowed and can be used with GAC-Schema with c-tuples.

For example, consider the ternary table constraint $C(V_1, V_2, V_3)$, with $D(V_1) = D(V_2) = D(V_3) = \{1, 2, 3\}$ that disallows the tuples $\langle 1, 2, 3\rangle$ and $\langle 3, 2, 1\rangle$. The optimal decision tree for this table is shown in figure 3. In this decision tree, the leaves that would normally contain the two tuples are instead labeled with the empty set, while the leaves that would be empty are used to construct c-tuples. The resulting set of allowed c-tuples is $\{\langle(1, 2, 3)(1, 3)(1, 2, 3)\rangle, \langle(1)(2)(1, 2)\rangle, \langle(2)(2)(1, 2, 3)\rangle, \langle(3)(2)(2, 3)\rangle\}$.

Fig. 3 Constructing a set of allowed c-tuples from a table constraint with forbidden tuples



Even though the number of allowed tuples may be exponentially larger than the number of forbidden tuples, the number of allowed compressed tuples will not be significantly bigger than the number of forbidden tuples.

Proposition 2. *Let C be a constraint on variables $V_1, \ldots, V_n$ with domains $D_1 = \ldots = D_n = D$ and $|D| = d$, represented as a set of forbidden tuples F. The size of the set C of compressed tuples generated from the empty leaves of the decision tree $T(F)$ is $O(nd|F|)$.*

Proof. Consider a leaf v that completely describes $F(n)$. Assume that at every node along this branch, following the alternative branch leads to an empty leaf. This is the maximum number of empty leaves that may correspond to each complete leaf. Since the maximum length of any branch is nd and there exist at most $|F|$ non-empty branches (in which case the set cannot be compressed), the maximum number of empty leaves is $O(nd|F|)$. □

Note that this upper bound is a worst case scenario, which assumes that the decision tree is maximal, thus the set of allowed u-tuples that represent the constraint cannot be compressed.

6 Empirical Results

We implemented the algorithm GAC3.1r [7] with and without compressed tuples. We compared the runtime for searching 100,000 nodes (for random problems)

or 1 million nodes (for the rest) for table constraints with compressed and uncompressed tuples, in the subset of families from section 3 where our heuristic algorithms were able to produce a smaller representation. We present our findings for some representative instances in table 2. We used the best splitting heuristic for each instance, as determined by the results in table 1.

Table 2. CPU time and number of constraint checks needed to search 100,000 nodes (for random problems) or 1,000,000 (for the `cril` instances) using both the uncompressed and compressed representation for table constraints, in instances from the 2005 CSP competition

						w/compression		w/out compression	
Instance	#Vars	#Cons	#Tables	t/t_c	l/l_c	Time	#CC×10^6	Time	#CC×10^6
random-3-20-20-60-632-forced-1	20	60	60	6.85	2.32	**76.26**	994.90	182.32	2010.40
random-3-20-20-60-632-forced-8	20	58	58	6.74	2.30	**73.48**	945.76	164.42	1935.04
random-3-20-20-60-632-forced-9	20	59	59	6.78	2.31	**120.65**	1165.17	234.57	2339.77
random-3-24-24-76-632-forced-1	24	74	74	8.15	2.40	**158.87**	1661.22	407.64	3207.54
random-3-24-24-76-632-forced-8	24	76	76	8.28	2.42	**207.80**	2033.63	503.94	4659.31
random-3-24-24-76-632-forced-9	24	74	74	8.12	2.40	**97.79**	1132.62	288.58	2922.41
random-3-28-28-93-632-forced-1	28	91	91	9.56	2.47	**268.93**	2495.32	739.16	6297.27
random-3-28-28-93-632-forced-8	28	92	92	9.67	2.48	**245.00**	2375.20	560.28	5470.00
random-3-28-28-93-632-forced-9	28	91	91	9.59	2.48	**494.69**	4543.70	1046.52	6440.60
cril_sat_nb_0	108	35	27	19.65	13.14	**22.57**	127.14	159.57	362.49
cril_unsat_nb_6	36	153	9	50.60	18.06	**10.99**	11.25	11.77	66.89
cril_unsat_nb_7	36	153	9	61.61	20.04	**19.21**	25.76	20.08	130.79

For each instance we present the number of variables, number of constraints, number of tables (which may differ from the number of constraints, as many constraint may share the same table,) the ratio t/t_c and l/l_c and finally the time needed to search 100,000 nodes and the number of constraint checks, in millions.

We see that in random problems using the compressed representation is uniformly better. The reduction in the number of constraint checks corresponds with the reduction in runtime over the simple version of the algorithm. In the `cril` instances, the picture is somewhat different. While the reduction in the number of constraint checks is significant, the difference in runtime does not reflect this. In the instance `cril_sat_nb_0`, the number of constraint checks is reduced by a factor of 13, but the runtime is only reduced by a factor of 7. In the other two instances, the number of constraint checks is reduced by a factor of 6, but the runtime is approximately the same with and without compression. Part of the reason for this is that these instances contain other constraints, so the improvement in speed is not apparent. For example, we found by profiling, that propagating the table constraints for the `cril_unsat_nb_6` only took approximately 15% of the total runtime for the uncompressed problem, therefore the improvement of .8 seconds actually corresponds to a 55% improvement in the time spent propagating the table constraints. This is still less than expected based on the number of constraint checks performed, however.

Finally, we also ran the `renault` configuration problem with both compressed and uncompressed tuples. In that problem, propagation takes too little time and thus in tests to find the first 2 million solutions, both algorithms performed identically. In profiling the programs, we found that propagation takes less 1%

of the total runtime and the majority of the time is instead spent on other aspects of backtracking search.

7 Related work

Our work on constructing small decision trees overlaps with similar work that has been performed in machine learning [10, 11]. As we mentioned earlier, the decision trees constructed for machine learning applications are intended to be used to classify as yet unseen tuples. Misclassification of some tuples is acceptable if it means keeping the size of the decision tree smaller. In our case, all tuples are known and we cannot accept any error in the classification.

GAC-Schema has been studied extensively. The work however has focused on more efficient search of the set of supporting tuples. Lhomme and Régin proposed the holotuple data structure [9] to avoid checking some tuples. Lecoutre and Szymanek proposed using binary search to locate a valid support [8]. Both these techniques are essentially orthogonal to using compressed tuples and can be used in conjunction with them.

In [4], it is proposed to use tries to represent the set of satisfying tuples. The tries can be viewed as a way to compress the shared prefixes of tuples. Even though tries have one leaf per tuple, finding a support may skip up to d^{n-m} invalid tuples if it encounters a pruned value at level $m < n$ of the trie. However, tries are restricted to having the same ordering along every branch, while our method of constructing decision trees is not restricted in this way. Moreover, in our method many branches may be combined (as we perform binary branching on the decision trees.) Thus, the reduction in the number of constraint checks that can be achieved using our method may be significantly better.

Finally, in [2] it is proposed to build a DAG where each node represents a range of values for a variable. A path from the root to a leaf in the DAG is equivalent to one c-tuple. However, no method is proposed to derive this DAG from an arbitrary set of tuples.

As far as we are aware, we are also the first to propose using this technique to propagate table constraints with sets of forbidden tuples.

8 Conclusions

We have presented an algorithm for compressing table constraints, for both the case when the table consists of allowed tuples and when the table consists of forbidden tuples. The representation produced contains c-tuples, each of which may correspond to exponentially many uncompressed tuples. Existing algorithms that enforced GAC on tables with allowed tuples can be adapted to work with c-tuples while maintaining the central structure of the algorithm that involves examining (c-)tuples for validity. As a result, compression can also deliver exponential time savings. Moreover, compression allows us to enforce GAC on tables with forbidden tuples in time polynomial in the number of forbidden tuples, while the best result so far has been exponential in the arity of the constraint. Finally, we demonstrated that this technique works in practice, using instances

from the 2005 CSP Competition. Moreover, instances where compression does not work can be detected relatively quickly.

Besides improving table constraint propagation, this work raises some questions. First, it appears that the heuristics developed in machine learning for creating small decision trees are not necessarily better than simpler alternatives. It would be interesting to develop better heuristics to create smaller trees. Alternatively, more effort could be put into finding better solutions to this NP-hard problem by utilizing constraint programming techniques. Finally, we intend to evaluate our methods on table constraints with forbidden tuples and integrate with other improvements to the GAC-Schema algorithm, such as the holotuples data structure of [9].

References

[1] C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 398–404, Nagoya, Japan, 1997.

[2] M. Carlsson. Filtering for the case constraint. Talk given at Advanced School on Global Constraints, Samos, Greece, 2006.

[3] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.

[4] I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty Second Conference on Artificial Intelligence (to appear)*, 2007.

[5] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[6] G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, 2005.

[7] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 125–130, 2007.

[8] C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming CP-06*, volume 4204 of *Lecture Notes in Computer Science*. Springer, 2006.

[9] O. Lhomme and J.-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005.

[10] J. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[11] J. Quinlan. *Programs for Machine Learning*. Morgan Kaurmann Publishers, 1993.

[12] J.-C. Régin. AC-*: A configurable, generic and adaptive arc consistency algorithm. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 505–519. Springer, 2005.

[13] M. van Dongen, C. Lecoutre, R. Wallace, and Y. Zhang. 2005 CSP solver competition. In M. van Dongen, editor, *"Second International Workshop on Constraint Propagation and Implementation"*, 2005.