Artificial Intelligence 220 (2015) 1–27

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

STR3: A path-optimal filtering algorithm for table constraints ☆

Christophe Lecoutre [a], Chavalit Likitvivatanavong [b,*], Roland H.C. Yap [b]

[a] CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France
[b] School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417, Singapore

ARTICLE INFO

Article history:
Received 27 August 2014
Received in revised form 3 December 2014
Accepted 9 December 2014
Available online 16 December 2014

Keywords:
Constraint satisfaction problems
Generalized arc consistency
Non-binary constraints
Backtracking search

ABSTRACT

Constraint propagation is a key to the success of Constraint Programming (CP). The principle is that filtering algorithms associated with constraints are executed in sequence until quiescence is reached. Many such algorithms have been proposed over the years to enforce the property called Generalized Arc Consistency (GAC) on many types of constraints, including table constraints that are defined extensionally. Recent advances in GAC algorithms for extensional constraints rely on directly manipulating tables during search. This is the case with a simple approach called Simple Tabular Reduction (STR), which systematically maintains tables of constraints to their relevant lists of tuples. In particular, STR2, a refined STR variant is among the most efficient GAC algorithms for positive table constraints. In this paper, we revisit this approach by proposing a new GAC algorithm called STR3 that is specifically designed to enforce GAC during backtrack search. By indexing tables and reasoning from deleted values, we show that STR3 can avoid systematically iterating over the full set of current tuples, contrary to STR2. An important property of STR3 is that it can completely avoid unnecessary traversal of tables, making it optimal along any path of the search tree. We also study a variant of STR3, based on an optimal circular way for traversing tables, and discuss the relationship of STR3 with two other optimal GAC algorithms introduced in the literature, namely, GAC4 and AC5TC-Tr. Finally, we demonstrate experimentally how STR3 is competitive with the state-of-the-art. In particular, our extensive experiments show that STR3 is generally faster than STR2 when the average size of tables is not reduced too drastically during search, making STR3 complementary to STR2.

1. Introduction

Algorithms that establish Generalized Arc Consistency (GAC) on constraint problems (networks) filter out inconsistent values from variable domains in order to reduce the combinatorial search spaces of such problems. They have been a staple of Constraint Programming (CP) since its origin in the field of Artificial Intelligence (AI) in the seventies, with for example the introduction of algorithms (G)AC3 [2] and (G)AC4 [3,4]. Typically, GAC is enforced at each step of a complete backtrack

☆ This is an expansion of the article which previously appeared in ECAI-12 [1].

* Corresponding author.
E-mail addresses: lecoutre@cril.fr (C. Lecoutre), likitchav@gmail.com (C. Likitvivatanavong), ryap@comp.nus.edu.sg (R.H.C. Yap).

http://dx.doi.org/10.1016/j.artint.2014.12.002
0004-3702/

	X	Y	Z
1	a	f	l
2	b	f	m
3	e	g	m
4	a	f	m
5	b	g	o
6	a	h	o
7	d	h	o
8	b	i	n
9	c	j	k

(a) Standard table

X		Y		Z	
a	{1,4,6}	f	{1,2,4}	k	{9}
b	{2,5,8}	g	{3,5}	l	{1}
c	{9}	h	{6,7}	m	{2,3,4}
d	{7}	i	{8}	n	{8}
e	{3}	j	{9}	o	{5,6,7}

(b) Dual table

Fig. 1. Standard and dual representations of the relation of a ternary constraint C with scope $\{X, Y, Z\}$.

search, leading to the so-called MAC, Maintaining (generalized) Arc Consistency, algorithm [5]. This paper introduces a new GAC algorithm, called STR3, that works with positive table constraints. Furthermore, unlike most GAC algorithms, STR3 is specifically conceived to be used within MAC rather than being standalone.

A table is just a relation, as in classical relational database, and a positive table constraint contains (in a table) all permitted combinations of values for a subset of variables (whereas a negative table constraint contains all forbidden combinations of values). Table constraints have been well studied in the artificial intelligence literature and arise naturally in many application areas. For example, in configuration and databases, they are introduced to model the problem no matter whatever the domain is. Besides, table constraints can be viewed as the universal mechanism for representing constraints, provided that space requirements can be controlled. The importance of table constraints makes them commonly implemented in all major constraint solvers that we are aware of (e.g., Choco, GeCode, JaCoP, OR-Tools).

For table constraints, many classical filtering algorithms that reduce search through inference (such as [6–9]) work with constraints that stay unaltered while running. However, recent developments suggested that reducing the amount of traversal by discarding irrelevant tuples from tables can lead to faster algorithms. Simple Tabular Reduction (STR) and its improvements [10,11] fall into this category and have been shown to be among the best GAC algorithms for positive table constraints.

The main idea behind simple tabular reduction is to remove invalid tuples from tables as soon as possible in a systematic fashion. STR3 is based on the same principle as STR1 [10] and STR2 [11] but employs a different representation of table constraints. Similarly to a few other algorithms (e.g., GAC-allowed [6] and GAC-va [8]), STR3 provides an index for each constraint table, enabling a tuple sought with respect to a domain value to be found without visiting irrelevant tuples, thus reducing time complexity. Fig. 1 shows an example for a ternary constraint. Importantly, for each constraint relation, STR3 maintains some specific data structures designed so that no constraint tuple is processed more than once along any path, through the search tree, going from the root to a leaf.

Most of the GAC algorithms for table constraints previously introduced in the literature suffer from repeatedly traversing the same tables or related data structures during search [11,12]. In contrast, STR3 avoids such repetition and is path-optimal: each element of a table is examined at most once along any path of the search tree. An important feature of STR3 is that it is designed specifically to be interleaved with backtracking search, where the main goal is to maintain the consistency while minimizing the cost of backtracking. As such, unlike most other GAC algorithms, STR3 is only applicable within the context of search: STR3 maintains GAC, but before commencement of search, GAC must be enforced by some other algorithm, such as STR2 for example.

We also investigated a promising circular manner for traversing tables in STR3. Although this seemed attractive at first because the circular approach described in [13] has an optimal run time per branch when amortized across a search tree, our experiments found that it was not really effective for STR3 in practice. To conclude our theoretical analysis, we discuss the relationships between STR3 and two other optimal GAC algorithms for table constraints, namely, GAC4 [4] and AC5TC-Tr [14].

We present an extensive experimental study that demonstrates that STR3 is competitive with state-of-the-art algorithms. In particular, our experiments show that STR3 is rather complementary to STR2. STR2 is faster than STR3 where simple tabular reduction can eliminate so many tuples from the tables that they become largely empty. STR3, by contrast, outperforms STR2 when constraint relations do not shrink very much during search (this is when STR2 is the more costly). Hence, STR3 is complementary to STR2.

This paper is organized as follows. Technical background is provided in Section 2. In Section 3, the concept of STR3 is explained together with its algorithm. A detailed example of STR3's step-by-step execution is given in Section 4. Section 5 analyzes the relationships among STR's data structures in greater detail. Theoretical analysis of STR3 is carried out in Sections 6 and 7. A variant of STR3 is studied in Section 8. Previous works related to STR3 are discussed in Section 9. Experimental results are reported in Section 10. The paper concludes in Section 11.

2. Preliminaries

In this section, we introduce some technical background concerning the constraint satisfaction problem, and we recall the operation of a data structure called sparse set, which is key to STR3's optimality.

2.1. Constraint satisfaction problem

A finite *constraint network* $\mathscr{P}$ is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ is a finite set of e constraints. Each *variable* $X \in \mathscr{X}$ has an *initial domain*, denoted by $D(X)$, which is the set of values that can be assigned to X. Each *constraint* $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$, denoted by $scp(C)$, that is called the *scope* of C. The *arity* of a constraint C is the number of variables involved in C, i.e., $|scp(C)|$. A *binary* constraint involves two variables whereas a *non-binary* constraint involves strictly more than two variables. The semantics of a constraint C is given by a *relation*, denoted by $rel(C)$; if $scp(C) = \{X_1, \ldots, X_r\}$, then $rel(C) \subseteq \prod_{i=1}^{r} D(X_i)$ represents the set of satisfying combinations of values, called *allowed* tuples, for the variables in $scp(C)$.

A *solution* to a constraint network is an assignment of a value to every variable such that every constraint is satisfied. A constraint network is *satisfiable* iff at least a solution exists. The *Constraint Satisfaction Problem* (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable or not. Thus, a CSP instance is defined by a constraint network, which is solved either by finding a solution or by proving that no solution exists. Solving a CSP instance usually involves a complete backtrack search that is interleaved with some inference processes to reduce the search space. In this paper, we shall focus on MAC (Maintaining Arc Consistency) [15], which is considered to be among the most efficient generic search algorithms for CSP. MAC explores the search space depth-first, backtracks when dead-ends occur, and enforces a property called (generalized) arc consistency after each decision (variable assignment or value refutation) taken during search.

Below, we introduce some notations and definitions that will be useful in the rest of the paper.

During search, $D^c(X)$ denotes the *current domain* of a variable $X \in \mathscr{X}$; we always have $D^c(X) \subseteq D(X)$. If a value $a \in D^c(X)$, we say that a is (currently) *present*; otherwise we say that a is *absent*. We use (X, a) to denote the value $a \in D(X)$ (or simply a when the context is clear), and we use $\tau[X_i]$ to denote the value a_i in any r-tuple $\tau = (a_1, \ldots, a_r)$ associated with a r-ary constraint C such that $scp(C) = \{X_1, \ldots, X_r\}$. A tuple $\tau \in rel(C)$ is *valid* iff $\tau[X] \in D^c(X)$ for each $X \in scp(C)$; otherwise τ is *invalid*. A tuple $\tau \in rel(C)$ is a *support* of (X, a) on C iff τ is valid and $\tau[X] = a$. We can now define *Generalized Arc Consistency* (GAC) as follows. A value (X, a) is generalized arc-consistent (GAC) on a constraint C involving X iff there exists a support of (X, a) on C. A constraint C is GAC iff for each $X \in scp(C)$ and for each $a \in D^c(X)$, (X, a) is GAC on C. A constraint network is GAC iff each of its constraints is GAC.

When $rel(C)$ is specified by enumerating its elements in a list, C is called a *positive table constraint*. Alternatively, $rel(C)$ may denote the set of forbidden combinations of values for the variables in $scp(C)$, in which case its extensional form is called a *negative table constraint*. A positive table constraint can be converted into a negative table constraint and vice versa. In this paper we deal only with positive table constraints. For any such constraint C, we assume a total ordering on $rel(C)$ and define $\texttt{pos}(C, \tau)$ to be the position of the tuple τ in that ordering (also called "tuple identifier" in the database literature, or *tid* for short [16,17]) whereas $\texttt{tup}(C, k)$ denotes the k-th tuple of $rel(C)$. Thus, $\tau = \texttt{tup}(C, \texttt{pos}(C, \tau))$ for any tuple $\tau \in rel(C)$, and $k = \texttt{pos}(C, \texttt{tup}(C, k))$ for any $k \in \{1, \ldots, t\}$ where $t = |rel(C)|$.

2.2. Sparse sets

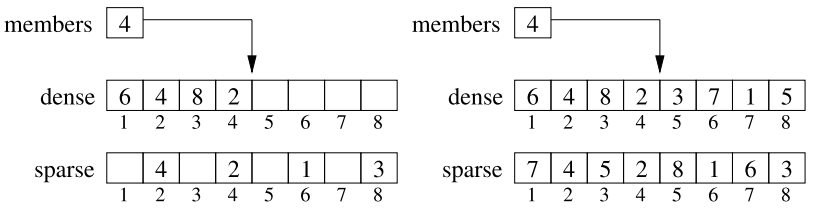
The sparse set data structure was first proposed in [18], with the motivation to provide fast operations on sets of objects. These operations are clear-set, insertion, deletion, membership test, and iteration. Sparse sets have played a crucial role in many recent CP algorithms [12,19,20]. In the context of backtracking search, we are concerned with the speed of only three basic operations, which are insertion, membership test, and deletion.

A sparse set S is an abstract structure composed of two arrays, traditionally called `dense` and `sparse`, together with an integer variable called `members`. Both arrays are of equal size n, with an index ranging from 1 to n, because possible elements are drawn from the universe $\{1, \ldots, n\}$. The array `dense` acts like a normal array container, with all elements packed at the left, while the array `sparse` carries each element present in the set to its location in `dense`. The value of `members` indicates the number of elements in S. Arrays `dense` and `sparse` adhere to the following property:

$$v \in S \quad \Leftrightarrow \quad S.\texttt{sparse}[v] \leq S.\texttt{members} \wedge S.\texttt{dense}\big[S.\texttt{sparse}[v]\big] = v. \tag{1}$$

An illustration is given by Fig. 2a, where the sparse set representation of a set currently containing 4 values (out of 8 possible ones) is shown. Note that the arrays `dense` and `sparse` require no initialization.[1] Pseudo-code for insertion and membership test is given in Fig. 3. Deletion in LIFO (Last In, First Out) order is achieved by decreasing the value of `members`.

[1] Actually, this is true provided that `sparse` only contains strictly positive values. In [18], it is assumed that indexing starts at 0 and `sparse` is an array of unsigned integers.



(a) Focusing on present elements. (b) Handling both present and absent elements.

Fig. 2. A sparse set S with 4 elements: $S = \{2, 4, 6, 8\}$.

```
1 addMember(S, v)
2     S.members++
3     S.dense[S.members] ← v
4     S.sparse[v] ← S.members

5 isMember(S, v)
6     i ← S.sparse[v]
7     return i ≤ S.members and S.dense[i] = v
```

Fig. 3. Original sparse set operations.

```
1 addMember(S, v)
2     S.members++
3     w ← S.dense[S.members]
4     i ← S.sparse[v]
5     S.dense[S.members] ← v
6     S.sparse[v] ← S.members
7     S.dense[i] ← w
8     S.sparse[w] ← i

9 isMember(S, v)
10    return S.sparse[v] ≤ S.members
```

Fig. 4. Sparse set operations optimized for membership testing.

In practice, membership tests may occur more frequently than insertions. In this case, a trade-off can be made so that the cost of testing the membership of a value v in a sparse set S is reduced to simply checking whether $S.\texttt{sparse}[v] \leq S.\texttt{members}$, at the expense of a more expensive insertion [20]. The idea is to have the array `dense` containing a permutation of the n values. Operations on the set maintain such a permutation by swapping elements in and out of S. Because both arrays `dense` and `sparse` are initially filled with values from 1 to n, there must be only a single copy of each value in `dense` at any time. The condition $S.\texttt{dense}[S.\texttt{sparse}[v]] = v$ is thus unnecessary. The pseudo-code of the modified `addMember` and `isMember` is given in Fig. 4. In addition, note that the enumeration of the elements that are not present in S is as simple as iterating from $\texttt{members} + 1$ to n in `dense`. An illustration is given by Fig. 2b.

3. STR3

This section introduces STR3, an algorithm based on simple tabular reduction for enforcing GAC on positive table constraints. GAC algorithms normally follow the same pattern: a domain value is proved to be consistent by producing a valid tuple containing that value (in the case of positive table constraints) or by producing some evidence from auxiliary structures, e.g., a path in case of BDDs (Boolean Decision Diagrams), MDDs (Multi-valued Decision Diagrams), or tries [12,21,22]. This is usually done by traversing these structures and running tests on each sub-structure. Reducing the amount of traversal has long been the focus of many works. For table constraints, optimization techniques include skipping over irrelevant rows or columns [7,8], or by compressing the tables themselves [12,21,22].

Simple Tabular Reduction (STR) [10], called STR1 in this paper, is a GAC algorithm that dynamically revises tables during search. While other GAC algorithms treat tables like fixed structures or resort to tackling comparable structures created from those tables, STR1 shows that handling tables directly during backtracking search is not as expensive as once thought. STR1 works by scanning each tuple one by one. If a tuple is invalid, it is removed and the table is contracted by one row

as a result. Otherwise, the tuple is valid and its components are therefore domain values that have been proved to have a support. STR1 then considers the tuple again to collect these values into designated sets (called $\texttt{gacValues}(X)$ and defined for every variable X in the scope of the constraint). When STR1 has finished going through the whole table, any value not present in those sets has no support and will be removed from its domain.

STR2 [11] provides two improvements to STR1. When a tuple τ is being inspected for validity, there is no need to check whether $\tau[X] \in D^c(X)$ if there has been no change to the domain of X since the last time STR2 was called on this constraint. In addition, in case τ is valid, when the algorithm goes through each component of τ, $\tau[X]$ is skipped over if it is known that $\texttt{gacValues}(X) = D^c(X)$ (i.e., every single value of $D^c(X)$ is already supported). The first improvement however involves additional data structures which must be maintained in order to deliver full optimization benefit. STR2 with these data structures maintained is called STR2+ [11]. Throughout this paper and especially in the experiments section, when we refer to STR2 we mean STR2+. STR2w [49] improves the lower bound of STR2 through redesign and incorporation of watched tuples.

Like STR1 and STR2, STR3 no longer examines a tuple once it has been recognized as invalid. Unlike STR1 and STR2, STR3 does not immediately discard invalid tuples from tables. Indeed, STR3 does not process tables directly, but instead employs indexes allowing rapid identification of all tuples containing a given value of a given variable. STR3 keeps a separate data structure that enables validity checks to be done in constant time, rather than revising indexes dynamically during search.

3.1. Structures and features

The main data structures used in STR3 are the following:

- For any $C \in \mathscr{C}$, $X \in scp(C)$, and $a \in D(X)$, we introduce $\texttt{table}(C, X, a)$, called *sub-table* of C for (X, a), as the set of *tids* for allowed tuples of C involving (X, a), i.e., $\{\texttt{pos}(C, \tau) \mid \tau \in rel(C) \wedge \tau[X] = a\}$. We implement $\texttt{table}(C, X, a)$ as a simple array, with indexing starting at 1; the i-th element of $\texttt{table}(C, X, a)$ is then denoted by $\texttt{table}(C, X, a)[i]$. We use an integer $\texttt{table}(C, X, a).\texttt{sep}$, whose value lies between 1 and $|\texttt{table}(C, X, a)|$, which we call the *separator* of $\texttt{table}(C, X, a)$. The separator is such that $\texttt{table}(C, X, a)[\texttt{table}(C, X, a).\texttt{sep}]$ is the position of the last known support of (X, a) on C. For the sake of brevity, we sometimes use $\texttt{table}(C, X, a)[\uparrow]$ instead of $\texttt{table}(C, X, a)[\texttt{table}(C, X, a).\texttt{sep}]$. The value of $\texttt{sep}$ is maintained during search, that is to say, subject to save and restore operations.
- For any $C \in \mathscr{C}$, we introduce $\texttt{inv}(C)$ as the set of *tids* for allowed invalid tuples of C, i.e., $\{\texttt{pos}(C, \tau) \mid \tau \in rel(C) \wedge \tau \text{ is invalid}\}$. We implement $\texttt{inv}(C)$ as a sparse set. The value of $\texttt{inv}(C).\texttt{members}$ is subject to save/restore operations during search.
- For any $C \in \mathscr{C}$ and $k \in \{1, \ldots, |rel(C)|\}$, we introduce $\texttt{dep}(C, k)$ as the *dependency list* associated with the k-th tuple of $rel(C)$. If $(X, a) \in \texttt{dep}(C, k)$, this means that the tuple $\tau = \texttt{tup}(C, k)$ provides the justification for a to be present in $D^c(X)$, i.e., (X, a) *depends* on τ. We implement $\texttt{dep}(C, k)$ as a simple array (with indexing starting at 1) since only sequential iterations and basic transfers are required. The dependency lists may be altered during search but they are not maintained (i.e., subject to save/restore operations).
- We introduce two stacks, denoted by $\texttt{stackS}$ (S stands for Separator) and $\texttt{stackI}$ (I stands for Invalid), that allow us to store values of the form $\texttt{table}(C, X, a).\texttt{sep}$ and $\texttt{inv}(C).\texttt{members}$ at different levels of search. Whenever a node in the search tree is visited for the first time, it is assumed that an empty container is placed on top of $\texttt{stackS}$ and $\texttt{stackI}$.

STR3 is a fine-grained filtering algorithm, meaning that the filtering process, called propagation, is guided by events corresponding to deleted values that are put in a so-called propagation queue. Here are some important attributes of STR3:

- When a value (X, a) is deleted, it is put in the propagation queue. This value is then picked from the queue later by the main constraint propagation procedure (not detailed in this paper), and STR3 is invoked for every constraint C such that $X \in scp(C)$. This invocation merges $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$, because $\texttt{table}(C, X, a)$ contains the *tids* of all tuples that involve the deleted value (X, a).
- Subsequently, STR3 recognizes that a value (Y, b) is GAC on C if $\texttt{table}(C, Y, b) \setminus \texttt{inv}(C)$ is not empty.
- The separator $\texttt{table}(C, X, a).\texttt{sep}$ is useful to distinguish between two regions: the explored region which contains *tids* of tuples (of $rel(C)$ involving (X, a)) known to be invalid, and the unexplored region which contain *tids* of tuples not yet examined. Each separator moves sequentially from one end of $\texttt{table}(C, X, a)$ to the other in a fixed direction. As search progresses, the explored region expands until it encompasses the whole set, at which point (X, a) has been proved not to be GAC on C.
- To check whether k is the *tid* of a tuple that has been found to be invalid, STR3 tests whether $k \in \texttt{inv}(C)$, through a call of the form $\texttt{isMember}(\texttt{inv}(C), k)$.
- Whenever a tuple of *tid* k becomes invalid, STR3 must look for a new support for every value in the dependency list $\texttt{dep}(C, k)$.

```
1  GACinit(C: Constraint)
2    remove invalid tuples from rel(C) and build every table(C,X,a)
3    inv(C).members ← 0
4    foreach k ∈ {1,...,|rel(C)|} do
5      dep(C,k) ← ∅
6    foreach X ∈ scp(C) and a ∈ D^c(X) do
7      table(C,X,a).sep ← |table(C,X,a)|
8      k ← table(C,X,a)[↑]
9      dep(C,k) ← dep(C,k) ∪ {(X,a)}

10 STR3(C: Constraint, X: Variable, a: Value)
11   membersBefore ← inv(C).members
12   for p ← 1 to table(C,X,a).sep do
13     k ← table(C,X,a)[p]
14     if ¬isMember(inv(C),k) then
15       addMember(inv(C),k)
16   if membersBefore = inv(C).members then
17     return true
18   save(C,membersBefore,stackI)
19   foreach i ∈ {membersBefore+1,...,inv(C).members} do
20     k ← inv(C).dense[i]
21     foreach (Y,b) ∈ dep(C,k) such that b ∈ D^c(Y) do
22       p ← table(C,Y,b).sep
23       while p > 0 and isMember(inv(C),table(C,Y,b)[p]) do
24         p ← p − 1
25       if p = 0 then
26         removeValue(C,Y,b)
27         if D^c(Y) = ∅ then return false
28       else
29         if p ≠ table(C,Y,b).sep then
30           save((C,Y,b),table(C,Y,b).sep,stackS)
31           table(C,Y,b).sep ← p
32         move (Y,b) from dep(C,k) to dep(C,table(C,Y,b)[p])
33   return true
```

Fig. 5. Algorithm STR3.

3.2. Algorithm

We emphasize that STR3's sole purpose is to maintain GAC during search [23,24]. It is not designed to establish GAC stand-alone. For that role, a separate GAC algorithm that is able to enforce GAC from scratch is required, and it must be invoked before the search commences, usually in the preprocessing stage.

Pseudo-code of STR3 is given in Figs. 5 and 6. `GACInit` (lines 1–9) is called first to remove all invalid tuples (by calling another GAC algorithm at line 2) and to initialize all data structures. In the beginning, $\texttt{inv}(C).\texttt{members}$ is set to zero as the remaining tuples are all valid (line 3), while the separator of each $\texttt{table}(C, X, a)$ is set to its last possible index (recall that we start indexing at 1). We also put each value (X, a) into an arbitrary dependency list.

During search, `STR3` (lines 10–33) is called for a constraint C every time a value a is removed from the domain of a variable X involved in C. Note that the instantiation of a variable X effectively invokes $\texttt{STR3}(C, X, a)$ for every value a that is present in the domain of X at the time of the assignment but which is not the value assigned to X. For each removed value (X, a), every tuple whose *tid* is in $\texttt{table}(C, X, a)$ becomes invalid. `STR3` then merges these *tids* into $\texttt{inv}(C)$ if they are not already present (lines 12–15). Values that need new supports are later processed (lines 19–32); we shall discuss this part of the algorithm in more details in Section 5. Upon backtracking, functions `restoreS` and `restoreI` are called so as to restore elements $\texttt{table}(C, X, a).\texttt{sep}$ and $\texttt{inv}(C).\texttt{members}$, through the use of the stacks `stackS` and `stackI`. Values are stored in these stacks at lines 18 and 30 by calling function `save`.

4. Illustration

In this section, we trace the execution of STR3 on the ternary positive table constraint C, with scope $\{X, Y, Z\}$, depicted in Fig. 1a. For each value in the table, its index or sub-table is given in Fig. 1b. After GAC preprocessing, separators and dependency lists are initialized as shown in Fig. 7. The symbol $\lhd_p$, where p is a number, points to the value whose position

```
1  save(key, newData, store)
2     if (key, oldData) ∈ top(store) for some oldData then
3        replace (key, oldData) with (key, newData)
4     else
5        insert (key, newData) to top(store)

6  restoreS()
7     list ← pop(stackS)
8     foreach ((C, X, a), p) ∈ list do table(C, X, a).sep ← p

9  restoreI()
10    list ← pop(stackI)
11    foreach (C, m) ∈ list do inv(C).members ← m

12 removeValue(C: Constraint, X: Variable, a: Value)
13    remove a from D^c(X)
14    add (C', X, a) to the propagation queue where C' ≠ C and X ∈ scp(C')
```

Fig. 6. Auxiliary functions of Algorithm STR3.

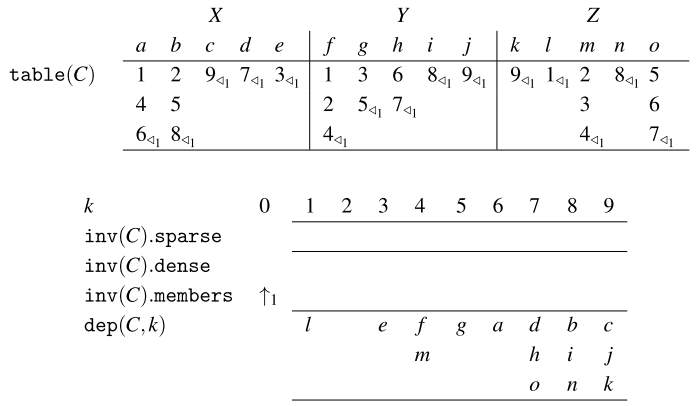


Fig. 7. Status right after GAC preprocessing at node η_1. Sub-tables and dependency lists are displayed vertically (at the top and the bottom of the figure, respectively).

is assigned to sep at node η_p. We also denote the fact that inv(C).members is assigned the value k at node η_p by placing $\uparrow_p$ at column k on the row titled inv(C).members. For instance at node η_1, we have inv(C).members $= 0$ and table(C, X, a).sep $= 3$ (because $|\texttt{table}(C, X, a)| = 3$ the value of the separator ranges from 1 to 3, from top to bottom). We can also observe for example that table(C, X, a) $= \{1, 4, 6\}$ and table(C, X, a)[3] $= 6$ (this will be always true since sub-tables are never directly modified; only the separators can change).

Assume values h, i, and o are eliminated. The result after STR3's propagation converges is shown in Fig. 8. In all figures hereafter, we mark a domain value that has been deleted by putting it inside a box. STR3 eventually merges the *tids* of tuples that involve these values (table(C, Y, h), table(C, Y, i), table(C, Z, o)) into inv(C), making inv(C).dense $= \{6, 7, 8, 5\}$. Values that depend on the tuples with those *tids* are in need of new supports. They are $\bigcup_{5 \le k \le 8} \texttt{dep}(C, k) \setminus \{h, i, o\} = \{g, a, d, b, n\}$ (values h, i, and o are already absent so they stay at their positions according to the condition in line 21 of STR3). Let us look first at what happens to g. STR3 locates a new valid support for g which is tup(C, 3). The value of table(C, Y, g).sep is then changed from 1 to 0 and g is transferred from dep(C, 5) to dep(C, 3). Similar operations are performed with respect to a and b. Now, concerning values d and n, it is clear that no other support exists. Consequently, these values can be deleted from their respective domains while continuing to exist in dep(C, 7) and dep(C, 8).

Now suppose that the search backtracks to η_1. The result is shown in Fig. 9. The dependency lists dep(C, k) are unaffected: they remain as in η_2, but note that they are different from those initially in η_1 (see Fig. 7 for comparison). This is an example of unsynchronized supports, which will be discussed in the next section. On the other hand, separators and inv(C).members have been rolled back to their previous values, as initially in η_1.

From this point, we suppose that e and n are eliminated. Fig. 10 shows the results after STR3 is called. The *tids* from both values' indexes (3 and 8) are first added to inv(C). Values for which STR3 needs to produce new supports are $\bigcup_{k \in \{3,8\}} \texttt{dep}(C, k) \setminus \{e, n\} = \{g, i\}$. Value i is removed because it has no further support. Value g stays present be-

	X					Y					Z				
	a	b	c	$\boxed{d}$	e	f	g	$\boxed{h}$	$\boxed{i}$	j	k	l	m	$\boxed{n}$	$\boxed{o}$
table(C)	1	$2_{\lhd_2}$	$9_{\lhd_1}$	$7_{\lhd_1}$	$3_{\lhd_1}$	1	$3_{\lhd_2}$	6	$8_{\lhd_1}$	$9_{\lhd_1}$	$9_{\lhd_1}$	$1_{\lhd_1}$	2	$8_{\lhd_1}$	5
	$4_{\lhd_2}$	5				2	$5_{\lhd_1}$	$7_{\lhd_1}$					3		6
	$6_{\lhd_1}$	$8_{\lhd_1}$				$4_{\lhd_1}$							$4_{\lhd_1}$		$7_{\lhd_1}$

k		0	1	2	3	4	5	6	7	8	9
inv(C).sparse							4	1	2	3	
inv(C).dense			6	7	8	5					
inv(C).members	$\uparrow_1$					$\uparrow_2$					
dep(C,k)			l	b	e	f			d	i	c
					g	m			h	n	j
						a			o		k

Fig. 8. Status at η_2.

	X					Y					Z				
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
table(C)	1	2	$9_{\lhd_1}$	$7_{\lhd_1}$	$3_{\lhd_1}$	1	3	6	$8_{\lhd_1}$	$9_{\lhd_1}$	$9_{\lhd_1}$	$1_{\lhd_1}$	2	$8_{\lhd_1}$	5
	4	5				2	$5_{\lhd_1}$	$7_{\lhd_1}$					3		6
	$6_{\lhd_1}$	$8_{\lhd_1}$				$4_{\lhd_1}$							$4_{\lhd_1}$		$7_{\lhd_1}$

k		0	1	2	3	4	5	6	7	8	9
inv(C).sparse							4	1	2	3	
inv(C).dense			6	7	8	5					
inv(C).members	$\uparrow_1$										
dep(C,k)			l	b	e	f			d	i	c
					g	m			h	n	j
						a			o		k

Fig. 9. After backtracking to η_1.

	X					Y					Z				
	a	b	c	d	$\boxed{e}$	f	g	h	$\boxed{i}$	j	k	l	m	$\boxed{n}$	o
table(C)	1	2	$9_{\lhd_1}$	$7_{\lhd_1}$	$3_{\lhd_1}$	1	3	6	$8_{\lhd_1}$	$9_{\lhd_1}$	$9_{\lhd_1}$	$1_{\lhd_1}$	2	$8_{\lhd_1}$	5
	4	5				2	$5_{\lhd_1}$	$7_{\lhd_1}$					3		6
	$6_{\lhd_1}$	$8_{\lhd_1}$				$4_{\lhd_1}$							$4_{\lhd_1}$		$7_{\lhd_1}$

k		0	1	2	3	4	5	6	7	8	9
inv(C).sparse					1		4	1	2	2	
inv(C).dense			3	8	8	5					
inv(C).members	$\uparrow_1$			$\uparrow_3$							
dep(C,k)			l	b	e	f	g		d	i	c
						m			h	n	j
						a			o		k

Fig. 10. Status at η_3.

cause $\texttt{tup}(C, \texttt{table}(C, Y, g)[\uparrow])$[2] $= \texttt{tup}(C, 5)$ is a support of g. STR3 then moves g from $\texttt{dep}(C, 3)$ to $\texttt{dep}(C, 5)$. Notice that the value of $\texttt{table}(C, Y, g).\texttt{sep}$ remains unchanged from Fig. 9. Finally, while it is true that the invalidation of $\texttt{table}(C, X, b)[\uparrow] = 8$ deprives b of a support, since b is not part of $\texttt{dep}(C, 8)$ STR3 will not try to find a new support for b. As a matter of fact, $b \in \texttt{dep}(C, 2)$. As a result, STR3 will start seeking a new support for b only after $\texttt{tup}(C, 2)$ becomes invalid.

[2] Again, $\texttt{table}(C, X, a)[\uparrow]$ refers to $\texttt{table}(C, X, a)[\texttt{table}(C, X, a).\texttt{sep}]$.

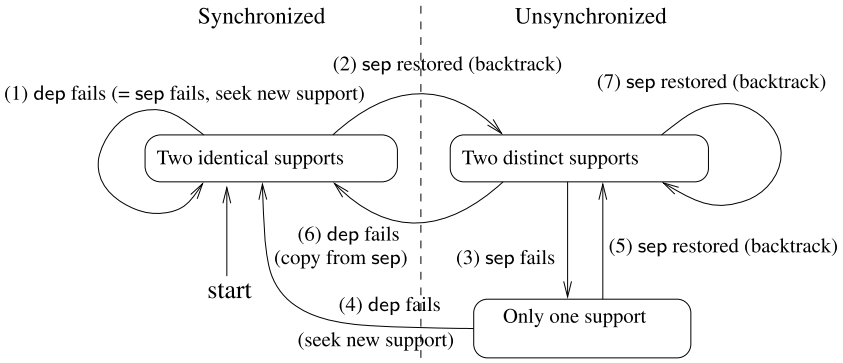


Fig. 11. Transition diagram depicting the relationship between the two supports of a value (X, a). STR3 is triggered only when dep fails. Both supports of (X, a) are synchronized as a result.

5. Synchronized vs. unsynchronized supports

Central to STR3 is the relationship between the separators and the dependency lists. A present value (X, a) is GAC on C because (1) $(X, a) \in \texttt{dep}(C, k)$ for some $k \notin \texttt{inv}(C)$, or (2) $\texttt{table}(C, X, a)[\uparrow] = k'$ with $k' \notin \texttt{inv}(C)$. Only one of the two conditions is sufficient for (X, a) to be proved to be GAC, and when both conditions are true, STR3 does not necessarily force the *tid* k to be equal to the *tid* k' as might be expected. This flexibility allows STR3 to keep a maximum of two different supports for each value with virtually no effort, thus almost doubling the chance that STR3 can avoid seeking a new support later. This section studies when and how this happens.

When $k = k'$, we say that the dependency lists and the separators are *synchronized* at (X, a) (or that the supports of (X, a) are synchronized). GACInit initializes separators and dependency lists so that they are synchronized from the beginning (see lines 6–9). In this case, the role of the dependency lists is straightforward: it just mirrors what happens to the separators. As soon as a tuple $\texttt{tup}(C, k)$ becomes invalid, STR3 looks for a new support for each value in the dependency list indexed at k, i.e., $\texttt{dep}(C, k)$ (line 21). Potential supports for a value (Y, b) are in the sub-table $\texttt{table}(C, Y, b)$, so, they are tested one by one against $\texttt{inv}(C)$, starting from the separator of the sub-table (lines 22–24). If no support is found the value is removed (line 26), and STR3 immediately fails when that value is the last one left in the domain (line 27). If a new support is found, the value of the current separator is recorded for backtrack purposes (line 30) before being replaced by the position of the new support (line 31). Dependency lists are always updated accordingly (line 32).

The separators and the dependency lists remain synchronized until backtracking occurs. Given $(X, a) \in \texttt{dep}(C, k)$ for some *tid* k, when the search backtracks $\texttt{dep}(C, k)$ remains unperturbed whereas $\texttt{table}(C, X, a).\texttt{sep}$ must revert back to its previous state if possible. For this reason, the separators and the dependency lists may no longer be synchronized at (X, a). In such cases, the tuples $\texttt{tup}(C, \texttt{table}(C, X, a)[\uparrow])$ and $\texttt{tup}(C, k)$ diverge and become two distinct supports of (X, a) on C.

We now consider in details different circumstances during STR3's execution when the validity of these two tuples later change (not including the cases where STR3 reports inconsistency). These relationships are portrayed in Fig. 11. We assume $|\texttt{table}(C, X, a)| > 1$ for all (X, a) so that the claim of two distinct supports is not trivially unfeasible. The diagram is explained in details as follows. We consider $(X, a) \in \texttt{dep}(C, k)$ for some *tid* $k \notin \texttt{inv}(C)$ and $\texttt{table}(C, X, a)[\uparrow] = k'$ for some *tid* $k' \notin \texttt{inv}(C)$. In Fig. 11, dep and sep failing means that k and k' are in $\texttt{inv}(C)$, respectively.

Supports are synchronized at the beginning, which means $k = k'$. There are two possible transitions:

(1) $\texttt{tup}(C, k)$ *becomes invalid*. STR3 must seek a new support in this case. Consequently, both supports remain synchronized (albeit with a *tid* different from k).
(2) $\texttt{table}(C, X, a).\texttt{sep}$ *is restored to some previous value*. This is caused by backtracking. The two supports of (X, a) become unsynchronized.

Suppose there are two distinct supports for (X, a) at this point. There are three possible transitions, namely (3), (6), and (7).

(3) $\texttt{tup}(C, k')$ *becomes invalid while* $\texttt{tup}(C, k)$ *remains valid.* Because STR3 seeks a new valid support for (X, a) only when $\texttt{tup}(C, k)$ is invalid (line 21), nothing needs to be done. This is what happens to value b in Fig. 10. There are two possible choices after this state:
  (4) $\texttt{tup}(C, k)$ *becomes invalid as well*. The search for a new valid support proceeds as usual. The dependency lists and the separators are synchronized at (X, a) on a newly acquired support if it exists.
  (5) $\texttt{table}(C, X, a).\texttt{sep}$ *is restored to some previous value*. Again, this is caused by backtracking and (X, a) would end up with two distinct supports as in case (2).

(6) $\texttt{tup}(C, k')$ *remains valid while* $\texttt{tup}(C, k)$ *becomes invalid.* (X, a) is simply shifted to another dependency list (i.e., from $\texttt{dep}(C, k)$ to $\texttt{dep}(C, k')$ by line 32). Because $\texttt{tup}(C, k')$ is valid, the second condition in line 23 will fail and the separator pointing at k' will never move. The dependency lists and the separators are synchronized at (X, a) as a result. In effect, the remaining support is copied over when the other fails. This is what happens to value g in Fig. 10.

(7) $\texttt{table}(C, X, a).\texttt{sep}$ *is restored to some previous value*. Same as (2) and (5).

Even though the trigger for STR3 to seek a new support is set up only on dependency lists, from the diagram it is clear that both $\texttt{tup}(C, k)$ and $\texttt{tup}(C, k')$ in fact provide two different sources of supports for (X, a); when one fails STR3 will draw on the other for support equally, and only after both fail does STR3 start seeking a new support. In an interesting scenario, a support in a dependency list may fail and get replaced by another support successively without STR3 having to explicitly seek a new support. The reason is that STR3 may cycle from synchronized to unsynchronized state and back repeatedly through cases (2) and (6), where the limit is the depth of the search tree.

While STR3 starts off with synchronized supports, unsynchronized supports are possible too. Indeed, because STR3 requires a form of preprocessing in order to remove invalid tuples, the ones that remain are all supports. Consider Fig. 1 for instance. After GAC preprocessing the tuples $\texttt{tup}(C, 2)$, $\texttt{tup}(C, 3)$, and $\texttt{tup}(C, 4)$ are all recognized as supports for (Z, m). In line 7, $\texttt{table}(C, Z, m).\texttt{sep}$ is assigned the value 3, therefore the ordinary STR3 that starts with synchronized supports would add (Z, m) to $\texttt{dep}(C, \texttt{table}(C, Z, m)[3]) = \texttt{dep}(C, 4)$. However, because the tuples $\texttt{tup}(C, 2)$, $\texttt{tup}(C, 3)$, and $\texttt{tup}(C, 4)$ are all supports of (Z, m), we could choose adding (Z, m) to $\texttt{dep}(C, \texttt{table}(C, Z, m)[1]) = \texttt{dep}(C, 2)$, or to $\texttt{dep}(C, \texttt{table}(C, Z, m)[2]) = \texttt{dep}(C, 3)$ as well. Since the separator moves down from the last cell to 1, choosing 1, the furthest cell away at the opposite end, would be the most natural choice. Line 8 would then be changed to:

$$k \leftarrow \texttt{table}(C, X, a)[1]$$

In our experiments, we use a variant of STR3 that starts with unsynchronized supports (thus benefiting initially from two supports for each value).

Observe that the separators and the dependency lists are somewhat comparable to watched literals [25] introduced for SAT. However, there are significant differences as follows. To begin with, for a given value, the relevant dependency list is the only activation point, working as a primary support while the separator serves as a possible backup; the separator points to a tuple that may or may not be valid. In contrast, there are always two watched literals for SAT, both equivalent in every way. For STR3, the separators are rigid and must maintain their values at all times while the dependency lists are not maintained, just like the two watched literals for SAT. Separators and dependency lists can be synchronized or unsynchronized depending on circumstances, in effect providing either a single support or two distinct supports whereas for SAT the two watched literals are always distinct where possible.

6. Correctness

This section proves properties that are crucial to STR3's correctness. An invariant is taken to be a property that remains true throughout the search, at the points before and after STR3 is enforced, but not necessarily while STR3 is running. For simplicity, arrays are sometimes perceived as mathematical sets in what follows.

The first invariant states that the dependency lists associated with a constraint C represent a disjoint collection of all domain values for the variables in the scope of C.

Invariant 1. *For any constraint* $C \in \mathscr{C}$*, the collection of sets* $\mathscr{S} := \{\texttt{dep}(C, i) : i \in 1 \dots |rel(C)|\}$ *represents a partition of* $\mathscr{D} := \bigcup_{X \in scp(C)} D(X)$ *such that every element of* $\mathscr{D}$ *is in one and only one set of* $\mathscr{S}$.

Proof. The invariant holds initially after GACInit is called. Line 32 contains the only statement that affects dependency lists and it moves one element from one list to another, thus preserving the invariant. □

The second invariant states that $\texttt{inv}(C)$ contains all *tids* of invalid tuples from $rel(C)$ and nothing else.

Invariant 2. *For any constraint* $C \in \mathscr{C}$, $\texttt{inv}(C) = \bigcup_{X \in scp(C), a \notin D^c(X)} \texttt{table}(C, X, a)$.

Proof. This is obvious from the fact that STR3 merges $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$ as soon as (X, a) becomes invalid, but it is worth emphasizing that there are exactly $|scp(C)|$ copies for each *tid* and they are distributed among different subtables (see Fig. 1b for example). Line 14 ensures that $\texttt{inv}(C)$ does not include duplicates and therefore conforms to the prerequisite of sparse sets. The invariant holds after backtracking because the right-hand side of the equation, $\bigcup_{X \in scp(C), a \notin D^c(X)} \texttt{table}(C, X, a)$, is conditioned upon domain values while the left-hand side, $\texttt{inv}(C)$, is controlled through $\texttt{inv}(C).\texttt{members}$, both of which are maintained by STR3. □

The third invariant guarantees that no support resides in the explored regions, i.e., after separators. We use $\texttt{table}(C, X, a).\texttt{explored}$ to denote $\texttt{table}(C, X, a)[\texttt{sep} + 1 \dots \texttt{size}]$ where $\texttt{sep} = \texttt{table}(C, X, a).\texttt{sep}$ and $\texttt{size} = |\texttt{table}(C, X, a)|$.

Invariant 3. *For any $C \in \mathscr{C}$, $X \in scp(C)$, and $a \in D^c(X)$, no* tid *of any support of (X, a) on C exists in* $\texttt{table}(C, X, a).\texttt{explored}$.

Proof. This invariant holds when the search starts since `GACInit` initially eliminates all invalid tuples and assigns the separators to their maximum values. Afterwards, when the tuple $\texttt{tup}(C, \texttt{table}(C, X, a)[\uparrow])$ becomes invalid, `STR3` scans down $\texttt{table}(C, X, a)$ until a new valid support is found, in which case the separator is set to the new value; the invariant holds. If no valid support is found, a is removed and the separator remains unchanged. The invariant still holds because it is conditioned on a being present in the domain. When a backtrack occurs, a becomes present again and the invariant still holds because the separator is maintained. □

Finally, the fourth invariant states that values stored in dependency lists do correspond to supports (for present values).

Invariant 4. *For any $C \in \mathscr{C}$, $X \in scp(C)$, and $a \in D^c(X)$, if $(X, a) \in \texttt{dep}(C, k)$, then $\texttt{tup}(C, k)$ is a support of (X, a) on C.*

Proof. The invariant holds right after `GACInit`. From the code, we see that whenever $\texttt{tup}(C, k)$ becomes invalid, any (X, a) in $\texttt{dep}(C, k)$ will be moved to another $\texttt{dep}(C, k')$ when a different valid *tid* k' is found (line 32). The invariants associated with $\texttt{dep}(C, k)$ and $\texttt{dep}(C, k')$ are preserved. If no valid *tid* is found, (X, a) becomes invalid. The invariant remains true because it deals only with present values.

We now look at the relationship between $\texttt{table}(C, X, a).\texttt{sep}$ and dependency lists when a backtrack is involved. If (X, a) switches from being absent to present after a backtrack, the invariant remains true, because either (1) (X, a) was removed as a consequence of the instantiation of X to some other value $b \neq a$, in which case the invariant is unaffected, or (2) chronological backtracking ensures that the tuple that (X, a) depended on most recently is restored as well (through rolling back of separators and $\texttt{inv}(C).\texttt{members}$). An interesting situation happens when (X, a) is present before and after backtracking. In this case, $\texttt{table}(C, X, a).\texttt{sep}$ may be reverted. Assume the value of $\texttt{table}(C, X, a)[\uparrow]$ is k before the backtrack, and k' after the backtrack (with $k < k'$). This means of course that (X, a) is in $\texttt{dep}(C, k)$ before the backtrack. Now, consider $\texttt{dep}(C, k)$ and $\texttt{dep}(C, k')$ after backtrack. Because backtracking never invalidates tuples, the tuple $\texttt{tup}(C, k)$ must still be valid after backtrack, and because dependency lists are not maintained, (X, a) remains in $\texttt{dep}(C, k)$. For this reason, the invariant for $\texttt{dep}(C, k)$ is still true, although $\texttt{table}(C, X, a)[\uparrow]$ is no longer k. The invariants involving values in $\texttt{dep}(C, k')$ are unaffected.

Next, consider what happens if the search moves forward when there are two distinct supports. That is, $(X, a) \in \texttt{dep}(C, k)$ while $\texttt{table}(C, X, a)[\uparrow] = k' \neq k$. If $\texttt{tup}(C, k)$ becomes invalid, we need to find a new support for (X, a). If there exists $1 \leq k'' \leq k'$ such that $\texttt{tup}(C, k'')$ is valid, `STR3` merely moves (X, a) from $\texttt{dep}(C, k)$ to $\texttt{dep}(C, k'')$. The invariants for $\texttt{dep}(C, k)$ and $\texttt{dep}(C, k'')$ hold afterward. If no valid support is found, (X, a) remains in $\texttt{dep}(C, k)$ and a becomes absent, making the invariant trivially true. □

We can now prove that STR3 does maintain GAC during backtrack search.

Theorem 1. *STR3 maintains GAC.*

Sketch of proof. We assume the standard value-based propagation framework and that the network is already GAC before STR3 is called for the first time. Two key observations for the completion of a proof are as follows. First, a value is deleted and put in the propagation queue as soon as STR3 exhausts all possibilities for supports (Invariant 3 where $\texttt{table}(C, X, a).\texttt{explored} = \texttt{table}(C, X, a)$ and line 26). Second, every present value has at least one valid support. This is due to Invariants 1 and 4 and the fact that a present value depends on exactly one tuple of a table constraint. □

7. Complexity

Many algorithms repeatedly compute a new value from an old one after a small modification to the computation context. An algorithm is incremental if it does not compute the new value from scratch but exploits both the old value and the modifications made to the environment. STR3 is designed to be incremental by avoiding repeated domain checks along the same path, going from the root to a leaf, in the search tree. In the following analysis, we consider the worst-case accumulated cost along a single path of length m in the search tree. It is assumed that (1) each variable domain is of size d, (2) each positive table constraint is of arity r and contains t tuples, (3) the tables do not contain invalid tuples before STR3 starts. Also, we consider that $d \leq t$ (a value $a \in D(X)$ must initially appear in all tables involving the variable X).

Theorem 2. *The worst-case space complexity of STR3 is $O(rt + mrd)$ per constraint.*

Proof. There are three main data structures in STR3: `table`, `inv` and `dep`. According to Invariant 1, the space complexity for `dep` is $O(rd)$. For `inv`, it is $O(t)$ whereas it is $O(rt)$ for `table`. For managing the restoration of data structures, we have `stackI`, which is $O(m)$, and `stackS`, which is $O(mrd)$, assuming that we may need to record information up to m levels. The total cost is $O(rd + t + t + rt + m + mrd)$, which is $O(rt + mrd)$. □

Theorem 3. *The worst-case time complexity of STR3 along a single path of length m in the search tree is $O(rt + m)$ per constraint.*

Proof. STR3's operations can be seen from the point of view of the three main data structures: `table`, `inv`, and `dep`. We consider them in this order:

- For a value a that stays present along a path, the cost of STR3 on $\texttt{table}(C, X, a)$ is $O(|\texttt{table}(C, X, a).\texttt{explored}|)$. If a is absent, there is an extra cost for merging the rest of $\texttt{table}(C, X, a)$ into $\texttt{inv}(C)$ (line 12), which is $O(|\texttt{table}(C, X, a)[1 \dots \texttt{sep}]|)$. In both cases, the cost is $O(|\texttt{table}(C, X, a)|)$. The total cost is $O(\sum_{X \in scp(C), a \in D(X)} |\texttt{table}(C, X, a)|) = O(rt)$.
- The maximum size of $\texttt{inv}(C)$ is t. Because $\texttt{inv}(C)$ is implemented as a sparse set, adding a member takes $O(1)$ time. The size of $\texttt{inv}(C)$ can only grow along the path so the cost of STR3 in dealing with $\texttt{inv}(C)$ is $O(t)$.
- Lastly, we consider the cost associated with dependency lists. When the tuple $\texttt{tup}(C, k)$ becomes invalid, each element in $\texttt{dep}(C, k)$ is processed and shifted if necessary. For each value (X, a), it can be shifted around at most $|\texttt{table}(C, X, a)|$ times. Because each dependency list $\texttt{dep}(C, k)$ is processed sequentially once along the path, the total cost is $O(\sum_{X \in scp(C), a \in D(X)} |\texttt{table}(C, X, a)|) = O(rt)$.

The worst-case time complexity of STR3 along a single path is thus $O(rt + t + rt + m) = O(rt + m)$, as all other statements have fixed costs ($O(1)$) at each node. □

As in the case of disjoint set union operations [26], a cascading series of transfers in dependency lists, where the number of elements to be moved keeps growing, is conceivable. This has no bearing on the complexity cost associated with dependency lists since we already show it to be $O(rt)$ through another argument, but it should be noted that the cascading cost is small due to the limited size of each list $\texttt{dep}(C, k)$, which holds no more than r elements. It follows that the worst-case cost in a monotonically increasing series of transfers is $O(\sum_{k=1}^{r-1} k) = O(r^2)$. Incidentally, the practical upper bound on operations involving dependency lists should be much closer to $O(rd)$ (the lower bound) than $O(rt)$ since absent values in dependency lists are never touched (line 21).

Property 1. *The worst-case time complexity along a single path of length m in the search tree can be as much as $O(rtm)$ for STR2 per constraint.*

Reasoning: Recall that STR2 improves over standard STR1 in two major ways. First, any (X, a) can be disregarded if $D^c(X)$ is fully supported. Second, no validity check is necessary for (X, a) if it is known that there is no change to the domain of X since the last time STR2 was called. Because STR2 is sensitive to ordering, we can build a table constraint and a search path such that (1) each call to STR2 involves a domain reduction of exactly one value on every domain, so that the second improvement is useless, (2) each call to STR2 eliminates exactly one tuple, which is found at the end of the table. As a result, the cost is $O(\sum_{i=1}^{m} r(t - i))$, which is $O(rtm)$ when $m \ll t$. □

It can be shown in a similar fashion that MDD^c [12] or tries [21] are not path-optimal. On the other hand, each backtrack costs $O(rd)$ in the worst-case for STR3, whereas it is $O(r)$ for STR2.

8. STR3 with circular seek

Gent [13] reported that seeking an element in an array in a circular fashion during backtracking search can be proved optimal when the cost is amortized. In other words, there is no theoretical difference with the traditional optimal procedure where a cursor's position is saved and restored upon backtrack. The circular approach is a factor of two slower in the worst case, but experiments with SAT solvers show that it can be faster in practice [13].

Because STR3 is based on incremental scans of the sub-tables, its cursors (separators) can be made to move circularly as well. The question is whether STR3 with circular seek remains optimal. This section addresses this variation of STR3 with respect to correctness, optimality, and performance.

Pseudo-code of STR3 with circular seek (STR3^{circ}) is given in Fig. 12. Differences from ordinary STR3 are:

d1 The search of supports performed from line 22 to 24 in Fig. 5 is modified to accommodate circular seek (line 13 to 21 in Fig. 12). The routine `restoreS` as well as the trailing of the separators at line 30 of Algorithm STR3 in Fig. 5 are no longer needed.

d2 The condition of the for-loop in line 12 of Algorithm STR3 in Fig. 5 is changed. The result is shown in line 3 in Fig. 12.

Theorem 4. STR3^{circ} *maintains GAC.*

Sketch of proof. STR3^{circ}'s correctness can be derived from STR3's when their differences are accounted for as follows:

Case d1: The circular move guarantees that, unless a variable X is assigned to a value $b \neq a$, the only condition for the value a to be removed is the absence of support in the sub-table involving (X, a). Because separators are not maintained,

```
1  STR3^circ(C: Constraint, X: Variable, a: Value)
2    membersBefore ← inv(C).members
3    for p ← 1 to |table(C,X,a)| do
4      k ← table(C,X,a)[p]
5      if ¬isMember(inv(C),k) then
6        addMember(inv(C),k)
7    if membersBefore = inv(C).members then
8      return true
9    save(C,membersBefore,stackI)
10   foreach i ∈ {membersBefore + 1,...,inv(C).members} do
11     k ← inv(C).dense[i]
12     foreach (Y,b) ∈ dep(C,k) such that b ∈ D^c(Y) do
13       p ← table(C,Y,b).sep
14       while p > 0 and isMember(inv(C),table(C,Y,b)[p]) do
15         p ← p − 1
16       if p = 0 then
17         p ← |table(C,Y,b)|
18         while p > table(C,Y,b).sep and
           isMember(inv(C),table(C,Y,b)[p]) do
19           p ← p − 1
20         if p = table(C,Y,b).sep then
21           p ← 0
22       if p = 0 then
23         removeValue(C,Y,b)
24         if D^c(Y) = ∅ then return false
25       else
26         if p ≠ table(C,Y,b).sep then
27           table(C,Y,b).sep ← p
28         move (Y,b) from dep(C,k) to dep(C,table(C,Y,b)[p])
29   return true
```

Fig. 12. STR3 with circular seek. Only the main procedure STR3^{circ} is shown.

it is always true that $(X,a) \in \texttt{dep}(C,k) \Leftrightarrow \texttt{table}(C,X,a).\texttt{sep} = k$. That is, separators and dependency lists are always synchronized. Invariant 4 holds as a result.

Case d2: In STR3, the subarray $\texttt{table}(C,X,a).\texttt{explored}$ always contains only *tids* of invalid tuples because the separators are not maintained. In STR3^{circ}, this is no longer true. Every *tid* in table has to be checked against $\texttt{inv}(C)$, but the effect on $\texttt{inv}(C)$ is the same as STR3's. □

Theorem 5. *The worst case time complexity of* STR3^{circ} *along a single path of length m in the search tree is* $O(rt + m)$ *per constraint.*

Proof. We need to look only at the operations involving table as the ones involving dep and inv are unchanged from STR3. For any value (X,a) that stays present along a single path, the cost of circular scan on $\texttt{table}(C,X,a)$ is $O(|\texttt{table}(C,X,a)|)$ according to Theorem 13 in [13]. If a is absent, there is an extra cost for merging $\texttt{table}(C,X,a)$ into $\texttt{inv}(C)$ (line 3 of Fig. 12). The cost is $O(|\texttt{table}(C,X,a)| + |\texttt{table}(C,X,a)|) = O(|\texttt{table}(C,X,a)|)$ for either case. Summing on X and a makes the final complexity $O(rt+m)$, where the factor m includes other $O(1)$ costs at each node. □

Proposition 14 in [13] indicates that overheads of the circular move can be a constant factor of two larger than the trailing of separators. For STR3^{circ}, there is a further cost of merging $\texttt{table}(C,X,a)$ for an absent value as mentioned in the proof above, which is another $O(rt)$ in total. Thus, the time complexity of STR3^{circ} hides a constant factor of three larger than STR3's. Our experiments show that STR3^{circ} is slower than both STR2 and STR3 on every problem instance tested.

9. Related works

A number of fine-grained (G)AC algorithms have been proposed in the literature, for example, AC5 [27], AC6 [28], AC7 [29], GAC4 [4], and AC5TC-Tr [14]. In this section, we discuss about the connections existing between GAC4, AC5TC-Tr, and STR3 since they are all optimal and closely related algorithms.

```
1  queue-processing(Q: Queue)
2    while Q ≠ ∅ do
3      pick and delete (X,a) from Q
4      foreach constraint C such that X ∈ scp(C) do
5        foreach tuple τ ∈ sup(C,X,a) do
6          foreach Y ∈ scp(C) | Y ≠ X do
7            b ← τ[Y]
8            remove τ from sup(C,Y,b)
9            if sup(C,Y,b) = ∅ and b ∈ D^c(Y) then
10             Q ← Q ∪ {(Y,b)}
11             D^c(Y) ← D^c(Y) \ {b}
```

Fig. 13. Algorithm GAC4.

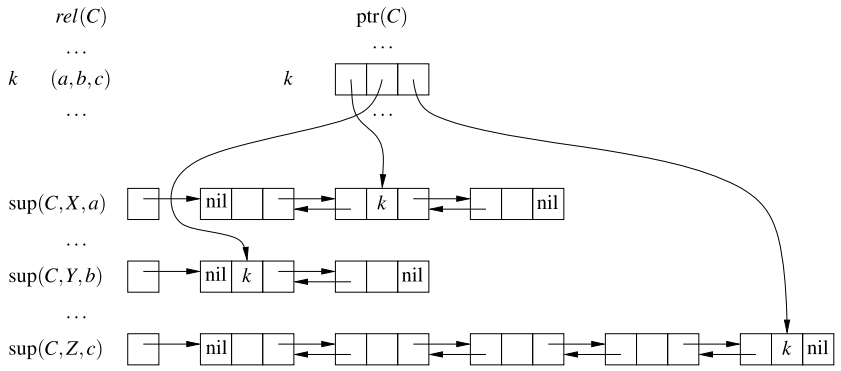


Fig. 14. Data structures of GAC4 for a constraint C such that $scp(C) = \{X, Y, Z\}$.

9.1. GAC4

GAC4 [4] is a generalized version of AC4 [3] for non-binary constraints. The pseudocode for the propagation portion of GAC4 is given by Fig. 13.

GAC4's approach to propagation is based on an incremental reduction of the support lists; the list of supports of a value (X, a) on a constraint C is denoted by $\text{sup}(C, X, a)$. When a value (X, a) becomes absent, every tuple τ involving that value becomes invalid. Because it may be a support for other values, $\text{sup}(C, Y, \tau[Y])$ for every $Y \neq X$ must be updated as well. A value (X, a) has no longer a support on a constraint C when $\text{sup}(C, X, a) = \emptyset$. It is then removed from $D^c(X)$ and put on the queue Q for further processing.

In order to keep the time complexity optimal, it is proposed in [4] that the list of supports be implemented as a doubly linked list. Removing an element of a support list therefore takes $O(1)$ time. The use of an additional data structure containing pointers to elements of support lists is suggested so that these elements can also be accessed in $O(1)$ time — specifically, a two-dimensional array $\texttt{ptr}(C)$ of size $|rel(C)| \times |scp(C)|$. For every tuple $\texttt{tup}(C, k)$, $1 \leq k \leq |rel(C)|$, $\texttt{ptr}(C)[k]$ is a one-dimensional array of size r, where $scp(C) = \{X_1, \ldots, X_r\}$, such that for all $1 \leq i \leq r$, if $\texttt{tup}(C, k)[X_i] = a$ then $\texttt{ptr}(C)[k][i]$ points to a node in $\text{sup}(C, X_i, a)$ whose value is k. Fig. 14 illustrates this with a ternary constraint C such that $scp(C) = \{X, Y, Z\}$.

Although attractive because admitting an optimal time complexity, GAC4 remains basically a standalone GAC algorithm. Indeed, no provision has been made for it to be used as a filtering step during backtracking search, although one can always make a simple adaptation by trailing all the relevant data structures, which are sup and ptr in this case. This incurs overheads, and it is unclear how costly the maintenance would be in practice since MGAC4 has never been reported in any experiment as far as we are aware. However, note that GAC4 remains useful when pre-caching of all supports is considered worthwhile. For instance, an extension of GAC4 is employed to find a form of interchangeable values in interactive configuration problems [30].

9.2. AC5TC-Tr

Several optimal GAC algorithms for table constraints are described in [14,31]. As a representative example, we focus our attention here on AC5TC-Tr [14] (see [31] for a comprehensive treatment of GAC algorithms for table constraints based on the AC5 structure [27]). We provide pseudocode of a key procedure of AC5TC-Tr in Fig. 15 and explain its function as follows.

```
1  ValRemoveTc-Tr(C: Constraint, X: Variable, a: Value)
2    △ ← ∅
3    i ← FS(C,X,a)
4    while i ≠ ⊤ do
5      foreach Y ∈ scp(C) such that Y ≠ X do
6        b ← tup(C,i)[Y]
7        if FS(C,Y,b) = i then
8          FS(C,Y,b) ← nextTr(C,Y,i)
9          if FS(C,Y,b) = ⊤ and b ∈ D(Y) then △ ← △ ∪ (Y,b)
10       else
11         if predTr(C,Y,i) ≠ ⊥ then
12           nextTr(C,Y,predTr(C,Y,i)) ← nextTr(C,Y,i)
13         if nextTr(C,Y,i) ≠ ⊤ then
14           predTr(C,Y,predTr(C,Y,i)) ← predTr(C,Y,i)
15     i ← nextTr(C,Y,i)
```

Fig. 15. Algorithm AC5TC-Tr.

In AC5TC-Tr, a support list on C for a value (X,a) is dynamically maintained as a doubly linked list headed by $FS(C,X,a)$ ("first support") while its elements are connected via pointers `nextTr` and `predTr`. Both pointers satisfy the following invariants for every variable $X \in scp(C)$ and every tuple $\texttt{tup}(C,i)$ where $1 \leq i \leq t = |rel(C)|$:

$$\begin{aligned}\texttt{nextTr}(C,X,i) = \min\big\{ j \mid i < j \wedge \texttt{tup}(C,j)[X] = \texttt{tup}(C,i)[X] \wedge \\ \big(\texttt{tup}(C,i) \in D(X,Q,C)\big) \Rightarrow \texttt{tup}(C,j) \in D(X,Q,C)\big\} \\ i = \texttt{predTr}\big(C,X,\texttt{nextTr}(C,X,i)\big)\end{aligned}$$

$D(X,Q,C)$ denotes the "local view" of the domain $D(X)$ with respect to the propagation queue Q, defined as $D(X,Q,C) = D(X) \cup \{a \mid (C,X,a) \in Q\}$. For any $\mathscr{V} \subseteq \mathscr{X}$, $D(\mathscr{V},Q,C) = \prod_{V \in \mathscr{X}} D(V,Q,C)$.

Furthermore, $FS(C,X,a)$, must satisfy the following invariant (a top value $\top$ is the largest value while a bottom $\bot$ is the smallest).

$$\begin{aligned}FS(C,X,a) = i \iff \texttt{tup}(C,i)[X] = a \wedge i \neq \top \wedge \texttt{tup}(C,i) \in D\big(scp(C),Q,C\big) \wedge \\ \forall j < i, \texttt{tup}(C,j)[X] = a \Rightarrow \texttt{tup}(C,j) \notin D\big(scp(C),Q,C\big)\end{aligned}$$

`valRemoveTC-Tr` is a primary function of AC5TC-Tr that deals with the consequences of the removal of a value (X,a) with respect to a constraint C. When (X,a) becomes absent, the function goes through each tuple τ in the support list of (X,a) and updates the support list of each value (Y,b) that contains τ. The idea is fundamentally the same as MGAC4, but with the operations on doubly linked lists spelled out. One important difference is that the data structure `ptr` is not used. Indeed, although the use of `ptr` was put forward in [4] it is actually not necessary because tuples are already grouped along domain values of a given variable. That is, assuming a total order on $rel(C)$, both $\texttt{nextTr}(C,X,i)$ and $\texttt{predTr}(C,X,i)$ return unique tuples (row positions). Thus next and previous tuples can be referred to directly through two pieces of information, namely, i (row) and X (column), as done in `valRemoveTC-Tr`.

STR3 and AC5TC-Tr achieve path-optimality through different routes. Both algorithms are based on the same concept of support lists, although STR3 works exclusively on row indexes. Traversals on the lists are analogous, whether through the doubly linked lists in AC5TC-Tr or through `table` in STR3. We discuss their differences in the rest of this section.

- *Indicators for the first valid supports.* The data structure *FS* plays the same role in AC5TC-Tr as `sep` does in STR3. Both mark the earliest valid support found according to the ordering in $rel(C)$. Due to its representation, STR3 needs `dep` as a reverse pointer in addition to `sep`. Because `dep` does not need trailing, STR3 is able to sidestep some of AC5TC-Tr's efforts. However this notion can be replicated in AC5TC-Tr by adding a similar data structure, which in this case serves purely as residues [32].
- *Overheads of doubly linked lists.* Besides the obvious extra traversal cost of doubly linked lists, the more important concern is its maintenance overheads during backtracks. AC5TC-Tr must keep track of the positions of all removed elements as well as the depth of the search when such removals happen. Should backtracks occur, AC5TC-Tr would use this information in order to restore these elements to their original positions. These removals may not be consecutive, and hence their restorations can be more expensive than the simple STR3's trailing mechanism.
- *Centralization of invalid tuples.* STR3 takes a lazy approach by partitioning a support list into two contiguous parts: an explored region known not to have any valid support, and an unexplored region. The two regions are separated by a cursor. As a result, when a tuple is proved invalid it must be remembered as such so that this fact can be recalled later when an unknown region is examined. To this end, STR3 channels all handling of invalid tuples through a centralized

facility (`inv`). By contrast, AC5TC-Tr actively maintains support lists: when a tuple becomes invalid it is immediately removed from all involving support lists so it will not be encountered again in the future.

- *Local views and granularity of invariants.* AC5TC-Tr is derived from the AC5 framework, whose correctness stems in turn from invariants on its data structures. These invariants differ from STR3's in two respects. First, in AC5 they deal with local views instead of the current domains. Invariants must therefore hold even for values that are absent but still remain in the local views; for instance the invariant on *FS* must be maintained even for some absent values. By contrast, STR3 suspends all operations on a value once it becomes absent. Second, AC5's invariants hold at the point before and after each value (X, a) is dequeued and processed. STR3's invariants, on the other hand, are coarser. During search they hold at the point before and after STR3 is completely executed, not during its execution where the filtering may not have yet converged to a fixed point.

Finally, STR3 can be regarded as an improved version of AC5TC-Sparse (previously called AC5TC-Cutoff in [14]) where an additional data structure is introduced for recording supports, which avoids restoring and checking operations to some extent.

10. Experimental results

In order to show the practical interest of STR3, we have conducted an experimentation (with our solver AbsCon) using a cluster of bi-quad cores Xeon processors at 2.66 GHz with 16 GB of RAM under Linux. We have compared STR3, first with other classical STR variants, and then with other related GAC algorithms developed for table constraints. For all our experiments, we have used the backtrack search algorithm called MAC, which maintains (G)AC during search, equipped with the variable ordering heuristic *dom/ddeg* [5] and the value ordering heuristic *lexico*. For each tested problem instance, we have searched to find a solution or prove that no solution exists, within 1200 seconds. It is important to note that the two chosen heuristics guarantee that we explore the very same search tree regardless of the filtering algorithm used (contrary to, for example, *dom/wdeg* [33]).

10.1. STR3 versus STR2: comparison on problem series

Because it has been shown that STR2 is state-of-the-art on many series of instances [11], we have compared the respective behavior of STR3 and STR2. Also included as a baseline are the results obtained with the original STR algorithm, as proposed by Ullmann [10] and referred to as STR1 here.

First, we have considered some classical series of instances[3] involving positive table constraints with arity greater than 2. We give a brief description of these series:

- The Crossword puzzle involves filling a blank grid using words from a specified dictionary. We have used four series of instances, called *crosswords-lex*, *crosswords-uk*, *crosswords-words* and *crosswords-ogd*, which have been generated from a set of grids without any black square, called *Vg*, and four dictionaries, called *lex*, *uk*, *words* and *ogd*. Dictionaries *lex* and *words* are small whereas *uk* and *ogd* are large. The arity of the constraints is given by the size of the grids: for example, *crosswords-ogd-5-6* involves table constraints of arity 5 and 6 (the grid being 5 by 6).
- A Renault Megane configuration problem, converted from symbolic domains to numeric ones, has been introduced in [34]. The series *renault-mod* contains instances generated from the original one, after introducing a form of perturbation. Such instances involve domains containing up to 42 values and some constraints of large arity (8 to 10); the largest table contains about 50 000 6-tuples.
- A Nonogram is built on a rectangular grid and requires filling in some of the squares in the unique feasible way according to some clues given on each row and column. Such clues can be modeled with table constraints. Constraint have typically large arities as for a grid of size $r \times c$, we get r constraints of arity c and c constraints of arity r. The size of the tables vary accordingly the size of the grids and the specified clues: some tables only contain a few tuples whereas the largest ones may usually contain tens or hundreds of thousands of tuples. The series *nonogram* here corresponds to the instances introduced in [35].
- Table constraints can be naturally derived from BDDs and MDDs. Series *bdd-15-21* and *bdd-18-21* were introduced in [22]: the former contains instances with table constraints of arity 15 and size 2713 whereas the latter contain instances with table constraints of arity 18 and size 133. Series *mdd-7-25-05* and *mdd-7-25-09* contain instances with constraints of arity 7 derived from MDDs built in a post-order manner with a specified probability p that controls how likely a previously created sub-MDD will be reused [12]. For the first series (also called *mdd-half*), the probability p is 0.5 whereas it is 0.9 for the second one.
- Series denoted by *rand-r-n* stand for random instances where each instance involves n variables and some constraints of arity r. The series *rand-3-20*, *rand-5-12*, *rand-8-20* and *rand-10-60* will permit us to experiment on random instances with various arity (from arity 3 to 10).

[3] Available at www.cril.univ-artois.fr/CSC09 or www.cril.fr/~lecoutre/benchmarks.html.

Table 1
Mean CPU time (in seconds) to solve instances from different series (a time-out of 1200 seconds was set per instance) with MAC.

Series	#		STR1	STR2	STR3
crosswords-lex	18	CPU	19.9	12.8	**12.6**
(avgP = 10.2% / avgS = 328)		mem	137M	137M	144M
crosswords-ogd	18	CPU	97.4	40.0	**25.0**
(avgP = 23.2% / avgS = 6070)		mem	123M	145M	236M
crosswords-uk	25	CPU	98.5	45.3	**43.1**
(avgP = 18.9% / avgS = 1500)		mem	126M	141M	183M
crosswords-words	23	CPU	62.8	37.4	**36.5**
(avgP = 11.9% / avgS = 637)		mem	138M	138M	150M
renault-mod	27	CPU	31.1	23.4	**20.6**
(avgP = 27.8% / avgS = 139)		mem	153M	153M	179M
nonogram	44	CPU	18.7	**9.4**	18.2
(avgP = 9.7% / avgS = 767)		mem	361M	386M	734M
bdd-15-21	35	CPU	64.0	**19.8**	60.7
(avgP = 6.7% / avgS = 466)		mem	219M	224M	2049M
bdd-18-21	35	CPU	23.9	**7.2**	142
(avgP = 6.1% / avgS = 3547)		mem	181M	182M	1043M
mdd-7-25-05	5	CPU	222	**130**	621
(avgP = 0.8% / avgS = 348)		mem	264M	265M	500M
mdd-7-25-09	9	CPU	154.0	**100**	105
(avgP = 5.9% / avgS = 2351)		mem	266M	267M	508M
rand-3-20	50	CPU	122	93	**79**
(avgP = 7.6% / avgS = 221)		mem	143M	143M	159M
rand-5-12	50	CPU	59.8	38.9	**15.1**
(avgP = 24.4% / avgS = 3048)		mem	259M	259M	485M
rand-8-20	18	CPU	25.2	**14.7**	24.8
(avgP = 0.2% / avgS = 191)		mem	221M	221M	379M
rand-10-60	19	CPU	352	191	**91.7**
(avgP = 23.0% / avgS = 11 750)		mem	248M	248M	457M

The results that we shall present include the following metrics:

- CPU time (in seconds); note that the CPU time for STR3 includes the preprocessing step (in which STR2 is employed).
- memory (mem) usage in MB.
- avgP (average proportion), which is the ratio "size of the current table" to "size of the initial table" averaged over all table constraints and over all nodes of the search tree; this is a relative value.
- avgS (average size), which is the size of the current table averaged over all table constraints and over all nodes of the search tree; this is an absolute value.

In order to avoid some "noise" generated by very easy instances, we have decided to discard from our results all instances that were systematically solved within 3 seconds (when embedding any filtering algorithm).

Table 1 shows mean results per series. Following the name of each series is the number of tested instances, which corresponds to the number of instances that are not too easy (as mentioned above) and not too difficult (solved by MAC within 1200 seconds when using any of the three algorithms). A first observation is that STR3 requires on average up to two times more memory than STR2; more memory was expected, but this is much better than what worst-case complexity suggests. A second observation is that the results seem to vary widely. STR2 and STR3 are respectively the best approaches on different series: *nonogram*, *bdd-15-21*, *bdd-18-21*, *mdd-7-25-05* and *rand-8-20* for STR2; *crosswords-ogd*, *rand-3-20*, *rand-5-12*, and *rand-10-60* for STR3. On other series, the gap between STR2 and STR3 is less significant. Table 2 gives details on some representative instances.

What is interesting to note, when looking at Tables 1 and 2, is that there appears to be a correlation between the values of avgP and avgS and the ranking of STR2 and STR3: the higher the values of avgP and avgS are, the more competitive STR3 becomes. Note that instances in Table 2 are ranked according to the values of avgP (from 0.5% to 51,2% for structured instances, and from 0.2% to 25, 6% for random instances) in order to make the transition more apparent. Intuitively, higher values of avgP and avgS also imply that there are fewer chances that the solver can reach deeper levels of the search tree, which in turn suggests a connection to unsatisfiability. To confirm this hypothesis, Table 3 divides crossword instances (all series taken together) according to satisfiability, an avgP threshold (pragmatically set to 10%) and avgS threshold (pragmatically set to 1000). Clearly, it appears that STR3 is the best approach when tables are not reduced too much in proportion and/or size (on average), contrary to STR2. For example, on the 46 Crossword instances for which avgS < 1000, STR2 is about 20% speedier than STR3 whereas on the 39 Crossword instances for which avgS ≥ 1000, STR3 is about 40% speedier than STR2.

Table 2
Detailed results on selected instances, sorted by avgP. The symbol "–" indicates a timeout.

		STR1	STR2	STR3
Structured instances				
crosswords-ogd-6-9	CPU	13.2	**7.9**	25.7
(sat – avgP = 0.5% – avgS = 227)	mem	148M	148M	202M
mdd-7-25-05-2	CPU	228	**123**	532
(sat – avgP = 1.2% – avgS = 467)	mem	264M	264M	500M
bdd-15-21-7	CPU	82.1	**24.5**	114
(unsat – avgP = 3.6% – avgS = 254)	mem	220M	225M	2063M
nonogram-143	CPU	11.7	**7.6**	18.5
(sat – avgP = 5.4% – avgS = 320)	mem	413M	422M	875M
crosswords-ogd-11-13	CPU	–	1086	**762**
(unsat – avgP = 10.7% – avgS = 5144)	mem	–	157M	294M
nonogram-65	CPU	64.6	17.3	**10.9**
(sat – avgP = 19.4% – avgS = 453)	mem	159M	163M	205M
crosswords-ogd-14-14	CPU	53.7	21.0	**12**
(unsat – avgP = 31.8% – avgS = 7491)	mem	144M	145M	236M
renault-27	CPU	21.4	15.6	**11.3**
(unsat – avgP = 51.2% – avgS = 223)	mem	151M	151M	178M
Random instances				
rand-8-20-8	CPU	86.2	**46.8**	565
(sat – avgP = 0.2% – avgS = 175)	mem	222M	222M	380M
rand-3-20-1	CPU	22.7	17.3	**17.1**
(sat – avgP = 4.6% – avgS = 136)	mem	144M	144M	161M
rand-3-20-26	CPU	243	178	**135**
(sat – avgP = 7.1% – avgS = 205)	mem	144M	144M	159M
rand-3-20-18	CPU	63.5	51.1	**38.2**
(unsat – avgP = 12.5% – avgS = 349)	mem	143M	143M	158M
rand-5-12-26	CPU	55.2	31.7	**13.3**
(unsat – avgP = 25.4% – avgS = 3167)	mem	259M	259M	486M
rand-10-60-5	CPU	319	114	**57.6**
(unsat – avgP = 25.6% – avgS = 13 141)	mem	248M	248M	457M

Table 3
Mean CPU time (in seconds) to solve Crossword instances (a time-out of 1200 seconds was set per instance) with MAC.

	#	STR1	STR2	STR3
sat	14	18.9	**11.5**	31.0
unsat	71	152	68.9	**52.8**
avgP $< 10\%$	34	93.9	**50.8**	56.7
avgP $\geq 10\%$	51	154	65.2	**44.3**
avgS < 1000	46	44.3	**26.7**	32.1
avgP ≥ 1000	39	231	98.0	**69.5**

Next, we tried to push the limit of the Crossword benchmarks by using a new dictionary *crosswords-ogd08*, which is twice larger than *ogd* (itself the largest one among the four dictionaries mentioned previously). There are 807 624 words in *ogd08* versus 435 705 in *ogd*. Most instances are of extreme cases: 45 instances are timed-out in all three algorithms tested and 12 are trivially solved (finished by both STR2 and STR3 within 3 seconds). The remaining instances are shown in Table 4 in order of the grid size (i.e. arity). Here we can see clearly the transition from satisfiable instances with low avgP and avgS values to unsatisfiable instances with high avgP and avgS values. STR2 is faster on the 4 first instances while STR3 is faster on the 4 last instances.

While it is not immediately clear what factors are involved concerning table reduction, one thing is certain: we know that every time a variable X is assigned a value a, all but tuples involving a are removed from the table of any constraint involving X, making avgP for this constraint low as a result. We have seen a surprising number of benchmarks with tables that are virtually wiped out by simple tabular reduction (no more than a few percent of the initial tuples remained on average) and variable instantiation may play an outsized role in this regard. We introduce now a benchmark where the instantiation is localized and has minimal effect on overall table reduction. A pigeonhole problem of size k is composed of k variables, each with $\{0, \ldots, k-1\}$ domain, and any two variables are connected by a binary inequality constraint, making the problem unsatisfiable. Unlike most benchmarks, the pigeonhole problem allows its tables to be reduced gradually during search until a variable directly involved is instantiated. An augmented pigeonhole ph-k-j adds an extra j-ary table and j new variables for each of the k variables and chain them together. The extra variables have larger domains to prevent the variable ordering heuristic from picking them prematurely (i.e., heuristics involving domain size) and the extra tables

Table 4
Results on the crossword puzzles for the *ogd08* dictionary. The symbol "–" indicates a timing out.

		STR1	STR2	STR3
crosswords-ogd08-6-8	CPU	2.14	**1.78**	5.59
(sat – avgP = 0.2% – avgS = 143)	mem	349M	349M	409M
crosswords-ogd08-6-9	CPU	12.8	**8.51**	51.8
(sat – avgP = 0.2% – avgS = 157)	mem	341M	341M	477M
crosswords-ogd08-7-7	CPU	2.41	**1.87**	4.94
(sat – avgP = 0.4% – avgS = 238)	mem	273M	273M	341M
crosswords-ogd08-7-8	CPU	–	**888**	–
(sat – avgP = 0.3% – avgS = 187)	mem	–	409M	–
crosswords-ogd08-15-17	CPU	–	1174	**735**
(unsat – avgP = 20.0% – avgS = 3657)	mem	–	341M	477M
crosswords-ogd08-15-19	CPU	414	152	**103**
(unsat – avgP = 24.9% – avgS = 3908)	mem	273M	273M	417M
crosswords-ogd08-16-19	CPU	778	346	**287**
(unsat – avgP = 14.4% – avgS = 1361)	mem	273M	273M	409M
crosswords-ogd08-16-20	CPU	226	96.4	**90.2**
(unsat – avgP = 19.2% – avgS = 1712)	mem	273M	273M	409M

Table 5
Results on the augmented pigeonhole problems.

		STR1	STR2	STR3
ph-6-9	CPU	20.3	10.6	**8.4**
(unsat – avgP = 65.2% – avgS = 1273K)	mem	551M	547M	1751M
ph-7-7	CPU	11.7	6.2	**4.8**
(unsat – avgP = 54.7% – avgS = 153K)	mem	65M	62M	290M
ph-7-8	CPU	73.9	37.6	**26.8**
(unsat – avgP = 54.7% – avgS = 919K)	mem	422	422M	1146M
ph-8-6	CPU	25.1	12.6	**5.1**
(unsat – avgP = 47.0% – avgS = 55 293)	mem	33M	33M	126M
ph-8-7	CPU	195.6	99.3	**71.1**
(unsat – avgP = 47.0% – avgS = 387K)	mem	207M	207M	631M
ph-9-5	CPU	44.3	23.4	**7.9**
(unsat – avgP = 41.1% – avgS = 13 479)	mem	18M	18M	40M
ph-9-6	CPU	389	201	**149**
(unsat- avgP = 41.1% – avgS = 108K)	mem	32M	32M	227M
ph-10-4	CPU	62.8	33.9	**17.3**
(unsat – avgP = 36.6% – avgS = 2399)	mem	15M	15M	21M
ph-11-3	CPU	86.4	63.8	**39.5**
(unsat – avgP = 32.9% – avgS = 329)	mem	10M	10M	21M

are very loose. Variable heuristics would be steered toward picking variables from the unsatisfiable core as a first priority. Results are shown in Table 5, where, once again, it is clear that STR3 is efficient with high values of avgP and avgS.

10.2. STR3 versus STR2: overall comparison

We present now a few scatter plots to provide an overall insight of the respective behaviors of STR2 and STR3. We use a large set of 2005 instances including the series introduced earlier, and also:

- the series of instances introduced in [31] that can be found at http://becool.info.ucl.ac.be/resources/,
- the series of instances introduced in [36],
- the series of instances introduced in [35] for problem *kakuro*.

In each plot, a dot represents an instance whose coordinates are defined by, on the horizontal axis, the CPU time required to solve the instance with STR2, and on the vertical axis, the CPU time required to solve the instance with STR3. Thus, every dot below the line $x = y$ corresponds to an instance solved more efficiently by STR3, and every dot above the line $x = y$ corresponds to an instance solved more efficiently by STR2.

Fig. 16 depicts with a first scatter plot the results obtained with STR2 and STR3 (within MAC) on the full set of 2005 instances. This scatter plot confirms that STR2 and STR3 are complementary: evidently, there are series/instances where STR2 is faster than STR3 and other ones where STR3 is faster than STR2. Figs. 17, 18 and 19 compare the performance of STR2 vs. STR3 with respect to satisfiability, value of avgP, and value of avgS. Finally, Fig. 20 plots the relative efficiency of STR2 against STR3 with respect to the value of avgS, when considering the 2005 instances of our experimental study. On some cases, where tables remain large (> 1000), STR3 can be up to $3.6\times$ faster than STR2.

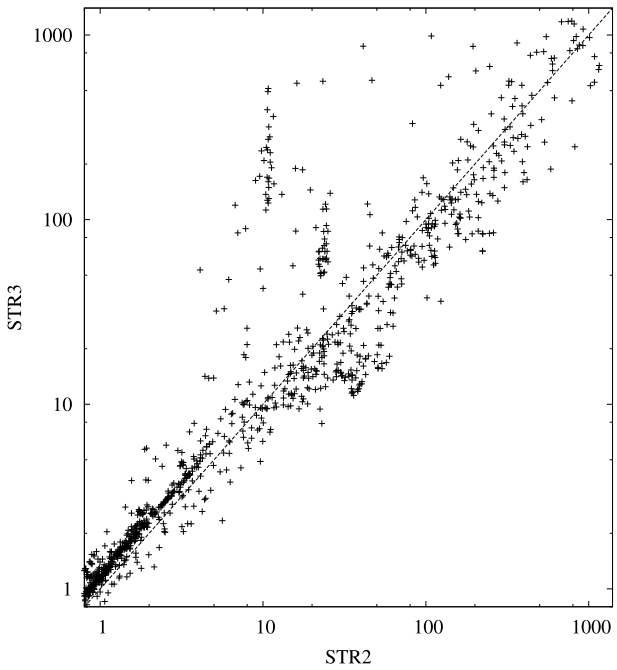


Fig. 16. Pairwise comparison (CPU time) on 2005 instances from many series involving table constraints. The time-out to solve an instance is 1200 seconds.

10.3. STR3 versus STR2: comparison on classes of random problems

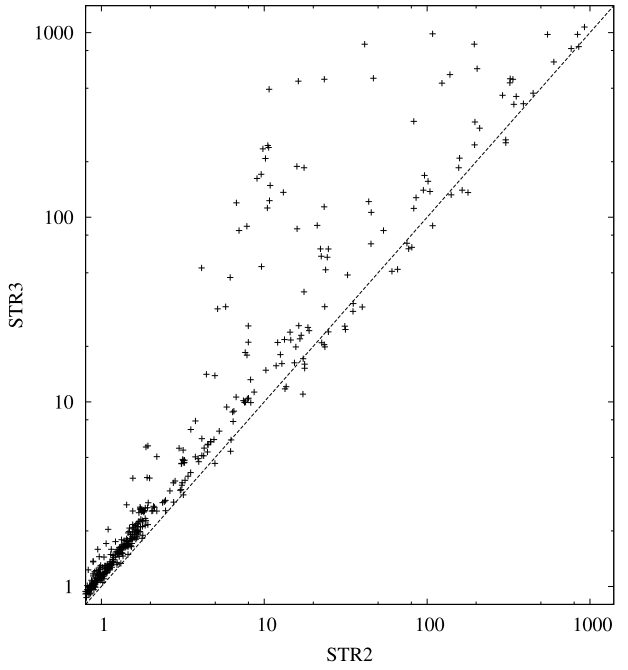
In the following set of experiments, we focus on classes of random problems, starting with those that can be found at the phase transition. We have generated different classes of instances from Model RD [37]. Each generated class $(r, 60, 2, 20, t)$ contains instances involving 60 Boolean variables and 20 r-ary constraints of tightness t. Provided that the arity r of the constraints is greater than or equal to 8, Theorem 2 [37] holds: an asymptotic phase transition is guaranteed at the threshold point $t_{cr} = 0.875$. It means that the hardest instances are generated when the tightness t is close to t_{cr}. Fig. 21a shows the mean CPU time required by MAC to solve 20 instances of each class $(13, 60, 2, 20, t)$ where t ranges from 0.8 to 0.96. On these instances of intermediate difficulty, we observe that STR3 is worse off than even STR1. Results on different classes of r follow the same pattern. Closer inspection reveals that avgP for this class is very low, especially at the phase transition where avgP is less than 4%.

For random problems, the metric avgP can be made higher when the instances that are generated do not lie in the phase transition area (therefore, there is no theoretical guarantee about their hardness). So, we have generated several classes of under-constrained instances. Each generated class $(5, 12, 12, 200, t)$ contains instances involving 12 variables with 12 possible values, and 200 constraints of arity 5 and tightness t. Fig. 21b shows the mean CPU time required by MAC to solve 10 instances of each class $(5, 12, 12, 200, t)$ where t ranges from 0.51 to 0.99; the size of the tables ranges from 121 928 (when $t = 0.51$) to 2488 (when $t = 0.99$). On these instances whose difficulty decreases with the tightness, we observe that STR3 is far better than STR2. Indeed, the values of both avgS and avgP are large: avgS ranges from 1093 to 8170 whereas avgP may reach up to 43%.

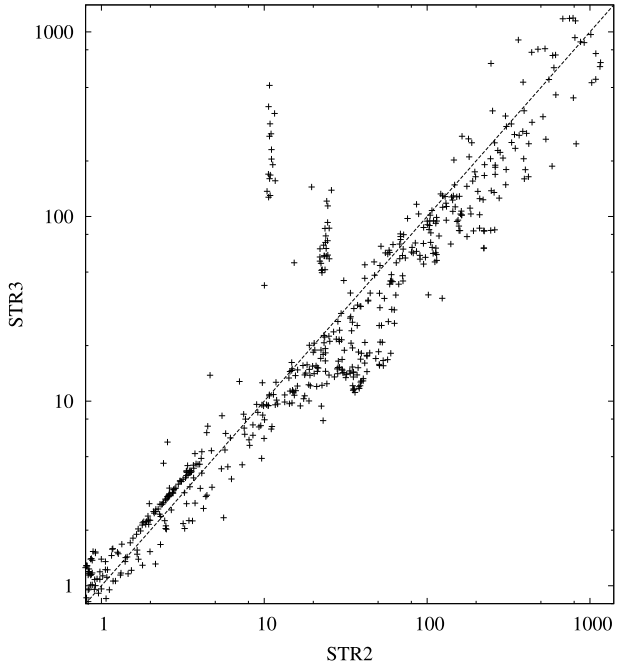
10.4. Comparison with other GAC algorithms

As mentioned in Section 9, STR3 and GAC4 are both optimal filtering algorithms. However, the overhead of maintaining the data structures of GAC4 during search can be significant. To confirm this, we have implemented GAC4 for it to be used within MAC by simply trailing its relevant data structures. Fig. 22 depicts with a scatter plot the results obtained with STR3 and GAC4 (within MAC) on the full set of 2005 instances involving table constraints. This scatter plot clearly shows that GAC4 is largely outperformed by STR3. Note that many crosses appear on the right of the figure, at $x = 1200$. They correspond to instances timed out by GAC4.

AC5TC, which has been proposed recently [31] is also an optimal algorithm. However, the solver AbsCon that we use does not permit, in its current shape, to implement easily this type of algorithms. This is the reason why we present in Table 6 an excerpt (with the kind permission of the authors) of the results obtained by Mairy, Van Hentenryck and Deville with the solver Comet. As mentioned in their paper (and can be observed from Table 6), STR2, STR3 and MDD^c outperform AC5TC when table constraints have large arity (i.e., greater than 4). However, on constraints of arity 3 and 4, AC5TC is shown to be very fast (see details in [31]).



(a) Comparison on satisfiable instances

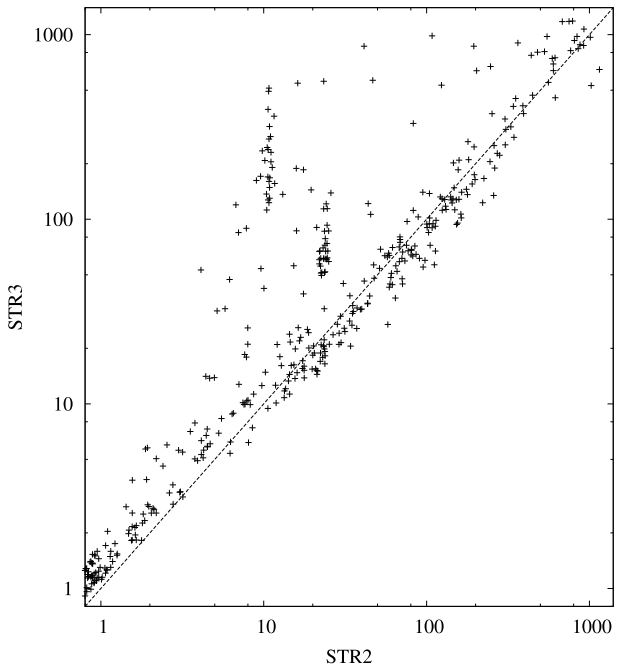


(b) Comparison on unsatisfiable instances

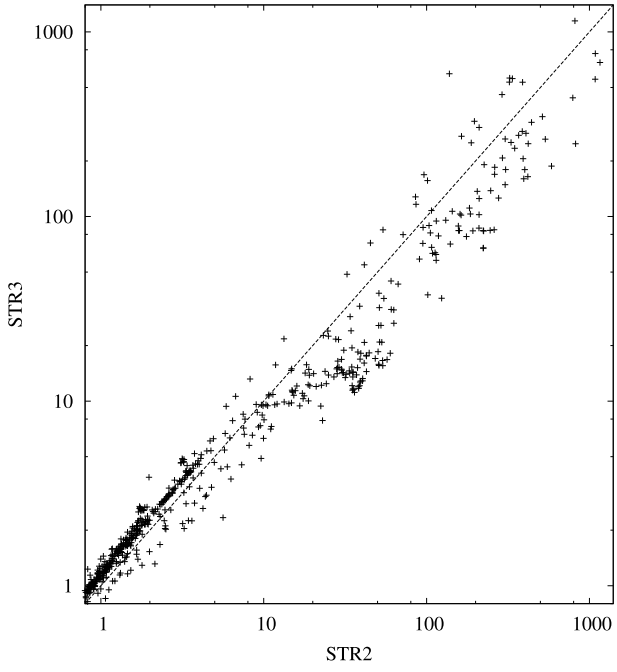
Fig. 17. Comparison (CPU time) of STR2 and STR3 on instances that are respectively satisfiable and unsatisfiable.

The reader must be aware that compression-based filtering algorithms remain appropriate when compression is highly effective. This has been shown in [38,22,39,12,36]. Typically, when the compression ratio is high, an algorithm such as MDD^c outperforms STR algorithms.

There are also three well-known binary encodings of non-binary constraint networks, called dual encoding [40], hidden variable encoding [41] and double encoding [42]. One could wonder whether or not a classical generic AC algorithm applied on such encodings could be competitive with simple tabular reduction. Actually, this has already been studied in [11], with



(a) Comparison on instances where avgP $< 10\%$

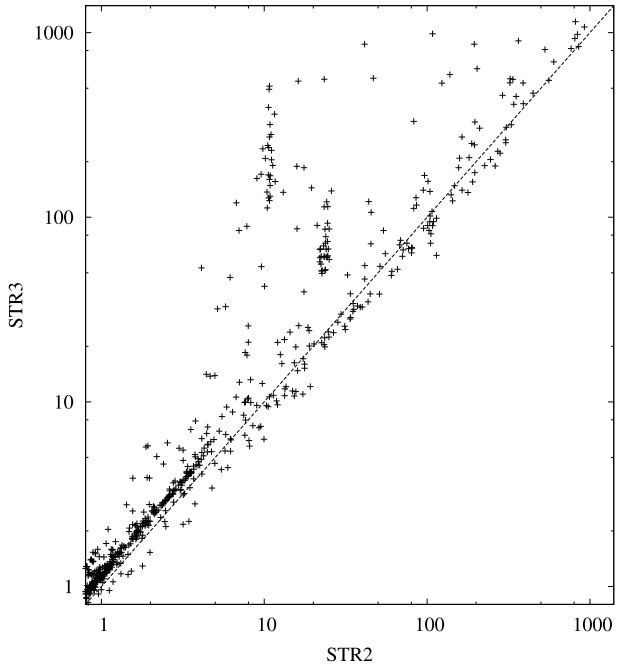


(b) Comparison on instances where avgP $\geq 10\%$

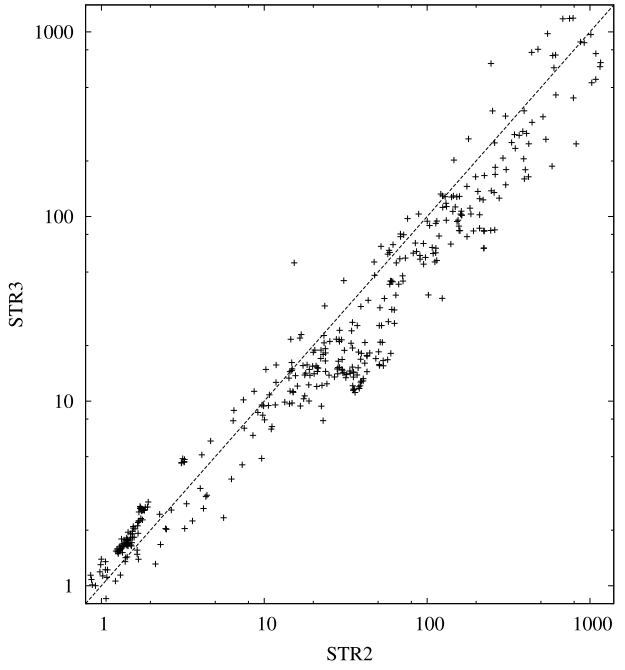
Fig. 18. Comparison (CPU time) of STR2 and STR3 on instances where avgP is respectively less than and greater than 10%.

respect to STR2: it is shown in that paper that the dual and the double encodings can rapidly run out of memory, and that STR2 is usually two or three times faster than AC3^{bit+rm} [43] and HAC [42] on the hidden variable encoding.

Finally, we would like to finish this presentation of experimental results with binary problems. Most of the recent works about filtering table constraints concentrate solely on non-binary constraints even though binary constraints in extensional form are tables too. Only after [31] was published that it was made clear the compression-based and the STR methods are not competitive with generic binary AC algorithms such as AC3^{rm} [44] and *AC3*$^{bit+rm}$ [43]. We confirm their findings with the results in Table 7.



(a) Comparison on instances where avgS < 1000



(b) Comparison on instances where avgS ≥ 1000

Fig. 19. Comparison (CPU time) of STR2 and STR3 on instances where avgS is respectively less than and greater than 1000.

11. Conclusions

We have introduced STR3, a new GAC algorithm for positive table constraints that is competitive and complementary to STR2, a state-of-the-art algorithm. STR3 is able to completely avoid unnecessary traversal of tables. Along with AC5TC [31], STR3 is one of the only two path-optimal GAC algorithms that have been reported so far. Unlike AC5TC's performance, which declines as arity increases, STR3's is consistent across a wide range of arity. Indeed, we have shown that it correlates to the average proportion (avgP) and number (avgS) of tuples remaining in tables during search. Compared to STR2,

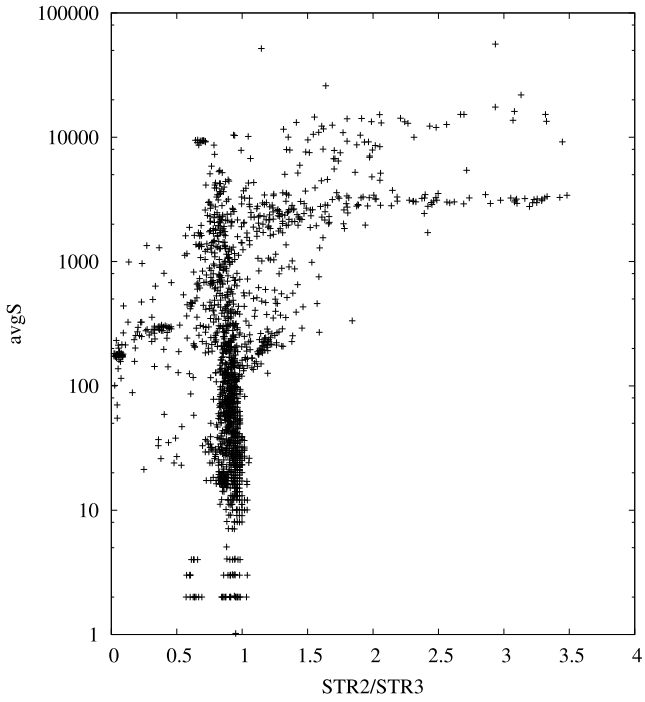


Fig. 20. The ratio "cpu STR2" to "cpu STR3" is plotted against avgS (average size of tables during search). Dots correspond to instances solved by both algorithms within MAC (from the set of 2005 instances used as our benchmark).

Table 6
Results obtained with AC5TC (version OptSparse) and MDD^c on classical series of instances involving table constraints of large arity. Values correspond to CPU times given as percentage to the best. This table is an excerpt from Table 10 in [31].

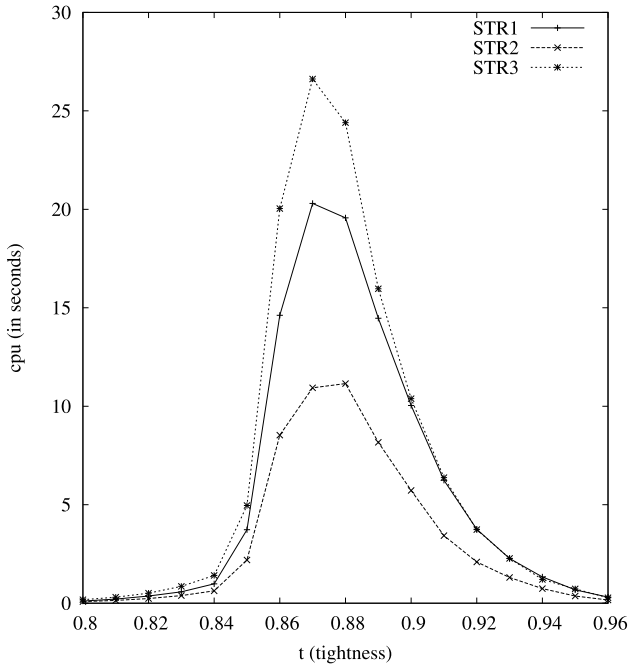
	AC5TC	MDD^c	STR2	STR3
crosswords-lex	116	293	121	**100**
crosswords-ogd	249	704	162	**100**
crosswords-uk	247	713	135	**100**
crosswords-words	155	328	138	**100**
renault-mod	141	332	**100**	193

Table 7
Mean CPU time (in seconds) to solve selected binary problems.

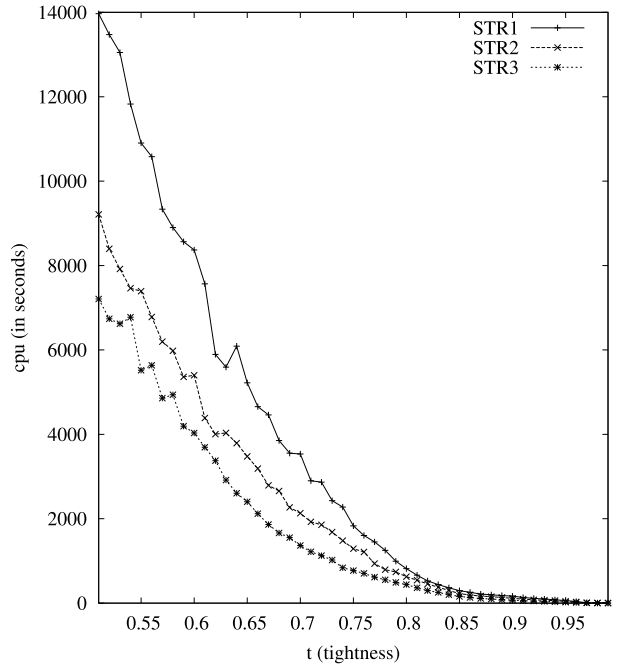
	#	AC3^{bit+rm}	AC3^{rm}	MDD^c	STR2	STR3
blackhole-4-4	10	**0.91**	1.10	1.24	1.42	1.81
bqwh-18-141	100	**17.9**	20.6	33.5	45.8	71.1
composed	14	**47**	62.3	105	119	164
driver	7	**7**	8.5	22.7	56.5	211
ehi-85	90	**47.7**	63.7	72	127	340
frb-45-21r	10	**91**	168	290	300	317
geom-50-20	100	**3.63**	6.7	12.3	12.6	14.1
qcp-10	10	**13.1**	15.8	29.3	43	72.5
qwh-15	10	**2.06**	2.52	3.94	5.49	14.1
rand-2-40	698	**5.42**	7.4	21.9	18.9	18.8

STR3 is faster on problems in which avgP and avgS are not low. Interestingly, the advantage of STR2 appears to depend largely on excessively high rates of table reduction (very low avgP). As soon as the reduction rate drops below 90%, STR2 becomes much less effective. Another dividing line is satisfiability: STR3 is stronger on unsatisfiable problems but weaker on satisfiable problems whereas STR2 is the opposite.

STR3 is an instance of fine-grained algorithms as its propagation is guided by deleted values. Both STR2 and STR3 are extended to handle compressed tuples (c-tuples) in [36]. While STR3 is more complex than STR2, once implemented the algorithm is easier to extend because it is based on just one notion: checking a tuple's validity; as a result, only the routine



(a) classes $(13, 60, 2, 20, t)$



(b) classes $(5, 12, 12, 200, t)$

Fig. 21. Mean search cost of solving instances in random classes with MAC.

involving validity test needs modification. By contrast, STR2 may process a tuple twice in different manners — testing its validity and then collecting values from its components. Extending STR2 to cope with c-tuples is therefore twice more complicated conceptually. This is true for fine-grained vs. coarse-grained propagation in general. In recent years, STR2 has been incorporated into many other algorithms [45–48], and it remains to be seen whether STR3 can be adopted in a similar fashion. The work which extends STR2 and STR3 to c-tuples [36] is a start in this direction.

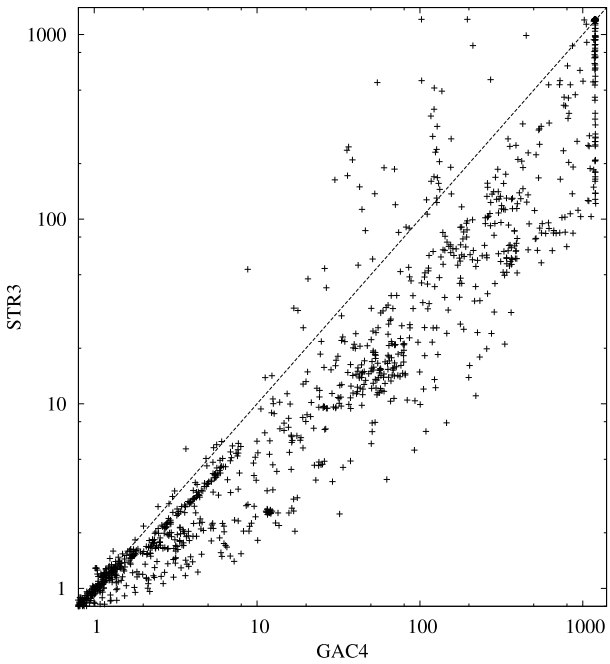


Fig. 22. Pairwise comparison (CPU time) on 2005 instances from many series involving table constraints. The time-out to solve an instance is 1200 seconds.

Acknowledgements

We thank Julian Ullmann for reading the manuscript and providing useful critical comments. This work has been supported by grant MOE2012-T2-1-155. The first author also benefits from the financial support of both CNRS and OSEO (BPI France) within the ISI project 'Pajero'.

References

[1] C. Lecoutre, C. Likitvivatanavong, R.H.C. Yap, A path-optimal GAC algorithm for table constraints, in: Proceedings of ECAI-12, Montpellier, France, 2012, pp. 510–515.
[2] A.K. Mackworth, Consistency in networks of relations, Artif. Intell. 8 (1) (1977) 99–118.
[3] R. Mohr, T.C. Henderson, Arc and path consistency revisited, Artif. Intell. 28 (2) (1986) 225–233.
[4] R. Mohr, G. Masini, Good old discrete relaxation, in: Proceedings of ECAI-88, Munich, Germany, 1988, pp. 651–656.
[5] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proceedings of CP'94, Orcas Island, USA, 1994, pp. 10–20.
[6] C. Bessière, J.-C. Régin, Arc consistency for general constraint networks: preliminary results, in: Proceedings of IJCAI-97, Nagoya, Japan, 1997, pp. 398–404.
[7] O. Lhomme, J.-C. Régin, A fast arc consistency algorithm for n-ary constraints, in: Proceedings of AAAI-05, Pittsburgh, Pennsylvania, 2005, pp. 405–410.
[8] C. Lecoutre, R. Szymanek, Generalized arc consistency for positive table constraints, in: Proceedings of CP-06, Nantes, France, 2006, pp. 284–298.
[9] C. Bessière, J.-C. Régin, R.H.C. Yap, Y. Zhang, An optimal coarse-grained arc consistency algorithm, Artif. Intell. 165 (2) (2005) 165–185.
[10] J.R. Ullmann, Partition search for non-binary constraint satisfaction, Inf. Sci. 177 (18) (2007) 3639–3678.
[11] C. Lecoutre, STR2: optimized simple tabular reduction for table constraints, Constraints 16 (4) (2011) 341–371.
[12] K.C.K. Cheng, R.H.C. Yap, An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints, Constraints 15 (2) (2010) 265–304.
[13] I.P. Gent, Optimal implementation of watched literals and more general techniques, J. Artif. Intell. Res. 48 (2013) 231–252.
[14] J.-B. Mairy, P.V. Hentenryck, Y. Deville, An optimal filtering algorithm for table constraints, in: Proceedings of CP-12, Quebec City, Canada, 2012, pp. 496–511.
[15] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proceedings of ECAI-94, Amsterdam, The Netherlands, 1994, pp. 125–129.
[16] M. Stonebraker, G. Held, E. Wong, P. Kreps, The design and implementation of INGRES, ACM Trans. Database Syst. 1 (3) (1976) 189–222.
[17] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, V. Watson, System R: relational approach to database management, ACM Trans. Database Syst. 1 (2) (1976) 97–137.
[18] P. Briggs, L. Torczon, An efficient representation for sparse sets, ACM Lett. Program. Lang. Syst. 2 (1–4) (1993) 59–69.
[19] V. le Clément de Saint-Marcq, P. Schauss, C. Solnon, C. Lecoutre, Sparse-sets for domain implementation, in: Proceedings of CP-13 Workshop on Techniques for Implementing Constraint Programming Systems, TRICS-13, Uppsala, Sweden, 2013.
[20] G. Gange, P.J. Stuckey, V. Lagoon, Fast set bounds propagation using a BDD-SAT hybrid, J. Artif. Intell. Res. 38 (2010) 307–338.
[21] I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Data structures for generalised arc consistency for extensional constraints, in: Proceedings of AAAI-07, Vancouver, Canada, 2007, pp. 191–197.
[22] K.C.K. Cheng, R.H.C. Yap, Maintaining generalized arc consistency on ad hoc n-ary boolean constraints, in: Proceedings of ECAI-06, Trento, Italy, 2006, pp. 78–82.
[23] C. Likitvivatanavong, Y. Zhang, S. Shannon, J. Bowen, E.C. Freuder, Arc consistency during search, in: Proceedings of IJCAI-07, Hyderabad, India, 2007, pp. 137–142.

[24] C. Likitvivatanavong, Y. Zhang, J. Bowen, E.C. Freuder, Arc consistency in MAC: a new perspective, in: Proceedings of CP-04 Workshop on Constraint Propagation and Implementation, CPAI-04, Toronto, Canada, 2004, pp. 93–108.
[25] M.W. Moskewisz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of DAC-2001, Las Vegas, NV, 2001, pp. 530–535.
[26] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third edition, The MIT Press, 2009.
[27] P.V. Hentenryck, Y. Deville, C. Teng, A generic arc-consistency algorithm and its specializations, Artif. Intell. 57 (2–3) (1992) 291–321.
[28] C. Bessière, M.-O. Cordier, Arc-consistency and arc-consistency again, in: Proceedings of AAAI-93, Washington, DC, 1993, pp. 108–113.
[29] C. Bessière, E.C. Freuder, J.-C. Régin, Using constraint metaknowledge to reduce arc consistency computation, Artif. Intell. 107 (1) (1999) 125–148.
[30] C. Becker, H. Fargier, Maintaining alternative values in constraint-based configuration, in: Proceedings of IJCAI-13, Beijing, China, 2013, pp. 454–460.
[31] J.-B. Mairy, P.V. Hentenryck, Y. Deville, Optimal and efficient filtering algorithms for table constraints, Constraints 19 (1) (2014) 77–120.
[32] C. Lecoutre, C. Likitvivatanavong, S.G. Shannon, R.H.C. Yap, Y. Zhang, Maintaining arc consistency with multiple residues, Constraint Program. Lett. 2 (2008) 3–19.
[33] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais, Boosting systematic search by weighting constraints, in: Proceedings of ECAI'04, Valencia, Spain, 2004, pp. 146–150.
[34] J. Amilhastre, H. Fargier, P. Marquis, Consistency restoration and explanations in dynamic CSPs — application to configuration, Artif. Intell. 135 (1–2) (2002) 199–234.
[35] G. Pesant, C.-G. Quimper, A. Zanarini, Counting-based search: branching heuristics for constraint satisfaction problems, J. Artif. Intell. Res. 43 (2012) 173–210.
[36] W. Xia, R.H.C. Yap, Optimizing STR algorithm with tuple compression, in: Proceedings of CP-13, Uppsala, Sweden, 2013, pp. 724–732.
[37] K. Xu, F. Boussemart, F. Hemery, C. Lecoutre, Random constraint satisfaction: easy generation of hard (satisfiable) instances, Artif. Intell. 171 (8–9) (2007) 514–534.
[38] G. Katsirelos, T. Walsh, A compression algorithm for large arity extensional constraints, in: Proceedings of CP-07, Providence, Rhode Island, 2007, pp. 379–393.
[39] K.C.K. Cheng, R.H.C. Yap, Maintaining generalized arc consistency on ad hoc r-ary constraints, in: Proceedings of CP-08, Sydney, Australia, 2008, pp. 509–523.
[40] R. Dechter, J. Pearl, Tree clustering for constraint networks, Artif. Intell. 38 (3) (1989) 353–366.
[41] F. Rossi, C. Petrie, V. Dhar, On the equivalence of constraint satisfaction problems, in: Proceedings of ECAI'90, Stockholm, Sweden, 1990, pp. 550–556.
[42] K. Stergiou, T. Walsh, Encodings of non-binary constraint satisfaction problems, in: Proceedings of AAAI'99, Orlando, Florida, 1999, pp. 163–168.
[43] C. Lecoutre, J. Vion, Enforcing arc consistency using bitwise operations, Constraint Program. Lett. 2 (2008) 21–35.
[44] C. Lecoutre, F. Hemery, A study of residual supports in arc consistency, in: Proceedings of IJCAI-07, Hyderabad, India, 2007, pp. 125–130.
[45] C. Lecoutre, A. Paparrizou, K. Stergiou, Extending STR to a higher-order consistency, in: Proceedings of AAAI-13, Washington, US, 2013, pp. 576–582.
[46] C. Jefferson, P. Nightingale, Extending simple tabular reduction with short supports, in: Proceedings of IJCAI-13, Beijing, China, 2013, pp. 573–579.
[47] C. Bessière, H. Fargier, C. Lecoutre, Global inverse consistency for interactive constraint satisfaction, in: Proceedings of CP-13, Uppsala, Sweden, 2013, pp. 159–174.
[48] N. Gharbi, F. Hemery, C. Lecoutre, O. Roussel, Sliced table constraints: combining compression and tabular reduction, in: Proceedings of CPAIOR-14, Cork, Ireland, 2014, pp. 120–135.
[49] C. Lecoutre, C. Likitvivatanavong, R.H.C. Yap, Improving the lower bound of simple tabular reduction, Constraints (2015).