Constraints manuscript No.
(will be inserted by the editor)

Optimal and Efficient Filtering Algorithms for Table Constraints

Jean-Baptiste Mairy · Pascal Van Hentenryck · Yves Deville

Abstract Filtering algorithms for table constraints can be classified in two categories: constraint-based and value-based. In the constraint-based approaches, the propagation queue only contains information on the constraints that must be reconsidered. For the value-based approaches, the propagation queue also contains information on the removed values. This paper proposes five efficient value-based algorithms for table constraints. Two of them (AC5TCOpt-Tr and AC5TCOpt-Sparse) are proved to have an optimal time complexity of $O(r \cdot t + r \cdot d)$ per table constraint. Substantial experimental results are presented. An empirical analysis is conducted on the effect of the arity of the tables. The experiments show that our propagators are the best when the arity of the table is 3 or 4. Indeed, on instances containing only binary constraints, our algorithms are outperformed by classical AC algorithm AC3rm. AC3rm is dedicated to binary constraints. However, all our algorithms outperform existing state-of-the-art constraint based STR2+ and MDD^c and the optimal value-based STR3 algorithms on those instances. On instances with small arity tables (up to arity 4), all our algorithms are generally faster than STR2+, MDD^c and than STR3. AC5TCOpt-Sparse is globally the best propagator on those benchmarks. On benchmarks containing large arity tables (arity 5 or more), each of the three existing state-of-the-art algorithms is the winning strategy on one different benchmark.

1 Introduction

Domain-consistency algorithms are usually classified as constraint-based (i.e., the propagation queue only contains information on the constraints that must be reconsidered) or value-based (i.e., removed values are also stored in the propagation queue). Table constraints

Yves Deville, Jean-Baptiste Mairy
ICTEAM, Université catholique de Louvain, Department of Computing Science and Engineering
Place Sainte-Barbe 2, 1348 Louvain-la-Neuve, Belgium
E-mail: {Yves.Deville,Jean-Baptiste.Mairy}@uclouvain.be

Pascal Van Hentenryck
NICTA and The University of Melbourne Lvl 2 / Bldg 193, The University of Melbourne Melbourne, VIC, Australia, 3010 E-mail: pvh@nicta.com.au

have been the focus of much research in recent years. Until recently, all existing algorithms (except in [21], [23] and [10]) were constraint-based. Recently, an optimal value based algorithm has been proposed in [19], together with an implementation. In this paper, we propose five original value-based algorithms for table constraints, which are all instances of the AC5 generic algorithm. The proposed propagators maintain, for every value of the variables, the index of its first current support in the table. They also use, for each variable of a tuple, the index of the next tuple sharing the same value for this variable. The algorithms differ in their use of information on the validity of the tuples and on the order of the tuples in the formed next chains. Those algorithms are similar to (G)AC4 [3, 23] propagators in the sense that they reason on the sets of supports of the literals. However, they differ from (G)AC4 by not explicitly representing nor maintaining the sets (except for AC5TCOpt-Sparse which uses efficiently backtrackable structures for it). Three of the proposed algorithms (AC5TC-Bool, AC5TC-Sparse and AC5TC-Recomp) have a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint and two of them (AC5TCOpt-Tr and AC5TCOpt-Sparse) have the optimal time complexity of $O(r \cdot t + r \cdot d)$, where r is the arity of the table, d the size of the largest domain and t the number of tuples in the table. One of the proposed algorithms, AC5TC-Recomp, is the (previously unpublished) propagator of the Comet system.

Experimental results show that the arity of the tables in the benchmarks has a substantial influence on the results of the propagators. We separate the benchmarks into three classes depending on the arity of the table constraints in the instances: binary constraints, small arity constraints (arity 3 and 4) and large arity constraints. On the problems containing only binary table constraints, our propagators are outperformed by the classical AC3rm algorithm [18]. AC3rm is designed only for binary constraints. However, on those instances, all our algorithms generally improve upon the existing state-of-the-art STR2+ [16], STR3 [19] and MDD^c [7]. On problems containing only small arity tables (up to arity 4), all our algorithms generally improve the existing state-of-the-art STR2+, STR3 and MDD^c: the speedup is up to 5.36 over STR2+, up to 7.82 over MDD^c and 3.25 over STR3. For the problems with larger arity tables (arity 5 or more), the conclusion is different. STR3, STR2+ and MDD^c are the winning strategy on one large arity benchmark each. Since the existing state-of-the-art propagators are suited for large arity tables, and that propagators for binary tables cannot be used for non-binary constraints, we expect our algorithms to be an interesting contribution to the field. The rest of this paper is organized as follows. Section 2 presents background information and related work. Section 3 describes the first two table constraint propagators. Section 4 proposes an efficient variant of our first algorithms. Section 5 presents our two optimal propagators and Section 6 describes the experimental results.

2 Background

A CSP $(X, D(X), C)$ is composed of a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $D(X) = \{D(x_1), \ldots, D(x_n)\}$ where $D(x)$ is the set of possible *values* for variable x, and a set of *constraints* $C = \{c_1, \ldots, c_e\}$, with $Vars(c_i) \subseteq X$ $(1 \leq i \leq e)$. We use $Vars(c)$ to refer to the scope of the constraint c. We will use $\#$ to refer to the cardinality of a set. We will say that a tuple $\mathbf{v}$ is in $D(X)$ iff $\forall 1 \leq i \leq n$, $\mathbf{v}_i \in D(x_i)$. $D(X)$ can thus be seen as the set of tuples $D(x_1) \times \ldots \times D(x_n)$. We let $d = max_{1 \leq i \leq n}(\#D(x_i))$, and $D(X)_{x_i=a}$ be the set of tuples $\mathbf{v}$ in $D(X)$ with $\mathbf{v}_i = a$. Given $Y = \{x_1, \ldots, x_k\} \subseteq X$, the set of tuples in $D(x_1) \times \ldots \times D(x_k)$ is denoted by $D(X)[Y]$ or simply $D(Y)$. We will

use the term literal to refer to a variable value pair. For a constraint c, we denote $c(\mathbf{v})$ the test evaluating to true iff $\mathbf{v}$ is allowed by c. By abuse of notation, when there is no ambiguity, $\mathbf{v}[x]$ will denote the value of variable x in tuple $\mathbf{v}$. A support in a constraint c for a literal (x, a) is a tuple $\mathbf{v} \in D(\mathit{Vars}(c))$ such that $c(\mathbf{v})$ and $\mathbf{v}[x] = a$. The following sets are useful for specifying domain consistency and propagation methods. Let c be a constraint with arity r, of a CSP $(X, D(X), C)$ with $y \in \mathit{Vars}(c)$, and $B(X)$ be some domain.

$$\begin{aligned} \mathit{Inc}(c, B(X)) &= \{(x,a) |\ x \in \mathit{Vars}(c) \wedge a \in D(x) \wedge \forall \mathbf{v} \in B(\mathit{Vars}(c))_{x=a} : \neg c(\mathbf{v})\} \\ \mathit{Cons}(c, y, b) &= \{(x,a) | x \in \mathit{Vars}(c) \wedge\ a \in D(x) \wedge \exists \mathbf{v} \in \mathbb{Z}^r : \mathbf{v}[x] = a \wedge \\ &\qquad\qquad \mathbf{v}[y] = b \wedge c(\mathbf{v})\} \\ \mathit{Inc}(c) &= \mathit{Inc}(c, D(X)) \end{aligned}$$

$\mathit{Inc}(c, B(X))$ represents the set of domain inconsistent literals of constraint c with respect to domain $B(X)$. $\mathit{Cons}(c, y, b)$ is the set of literals in the tuples allowed by c having value b for variable y. A constraint c in a CSP $(X, D(X), C)$ is *domain-consistent* iff $\mathit{Inc}(c) = \emptyset$. A CSP $(X, D(X), C)$ is *domain-consistent* iff all its constraints are domain-consistent.

Table Constraints. Given a set of tuples T of arity r, a table constraint c over T holds if $(x_1, \ldots, x_r) \in T$, where $\mathit{Vars}(c) = (x_1, \ldots, x_r)$. The size t of a table constraint c is its number of tuples, which is also denoted by $c.\mathit{length}$. We assume an implicit ordering of the tuples: $\sigma_{c,i}$ denotes the i^{th} element of the table in c and $\sigma_{c,i}[x]$ is the value of $\sigma_{c,i}$ for variable x. We introduce a top index $\top$ (resp. bottom index $\bot$) greater (resp. smaller) than any other index. We also introduce a universal tuple $\sigma_{c,\top}$, with $\sigma_{c,\top}[x] = *$ forall $x \in X$ and abuse notations in postulating that $\forall a \in D(x), * = a$. This universal tuple can thus be found in any table. More precisely, for any table T, $\sigma_{c,\top} \in T$. Given a table constraint, we say that a tuple σ is *allowed* if it belongs to the table. A tuple σ is *valid* if all its values belong to the domain of the corresponding variables. To achieve domain consistency, one must at least check the validity of each tuple and, in the worst case, remove all the values from the domains. Hence a domain-consistency algorithm has a complexity $\Omega(r \cdot t + r \cdot d)$ per table constraint in the worst case. An AC5-like algorithm with a complexity $O(r \cdot t + r \cdot d)$ per table constraint is thus optimal. For most incremental propagators, if the time complexity for obtaining domain consistency is $O(f)$, then the time complexity of the aggregate executions of this algorithm along any path in the search tree is also $O(f)$.

Related Work. This paper is an extended version of [22]. A lot of research effort has been spent on table constraints. The existing propagators can be categorized in 4 classes: index-based, compression-based, based on a dynamic table and value-based. Those classes are not mutually exclusive. For the presented approaches, we will analyze the complexity of all the calls to the propagator along a branch of the search tree. The index-based approaches use an indexing of the table to speed up its traversal. Examples of such propagators are GAC3-allowed and other constraint-based variants (GAC3$_{rm}$-allowed, GAC2001-allowed) [15, 4, 17, 11]. For each variable value pair (x, a), the index data structure has an array of the indexes of the tuples with value a for x. The space complexity of the data structure is $O(r \cdot t)$. The time complexity of GAC3-allowed along a branch in the search tree is $O(r^3 \cdot d \cdot t)$ per table constraint. Indeed, for one call to GAC3-allowed, a variable requires at worst to test the validity of each tuple of the constraint (thanks to the indexing structure), which costs $O(r \cdot t)$. Thus, one call of GAC3-allowed costs $O(r^2 \cdot t)$. The propagator

being called at most $r \cdot d$ times along a branch in the search tree, the complexity of GAC3-allowed is $O(r^3 \cdot d \cdot t)$. GAC2001-allowed has a time complexity of $O(r^2 \cdot t + r^3 \cdot d^2)$ per table constraint along a branch in the search tree. The last support structure of GAC2001-allowed ensures that a tuple from the table is never considered twice, except while checking if the last support is still valid. Along a branch in the search tree, the cost of testing each tuple once per variable is $O(r^2 \cdot t)$. The cost of the validity tests of the last supports is $O(r^3 \cdot d^2)$, leading to a total complexity of $O(r^2 \cdot t + r^3 \cdot d^2)$. Indexing can also be used in value-based propagators. In [21], the authors propose a value-based propagator for table constraints implementing GAC6. It uses a structure which indexes, for each variable value pair (x, a) and each tuple, the next tuple in the table with value a for x. The space complexity of the data structure is $O(r \cdot d \cdot t)$. This space usage can be reduced by using a data structure called hologram [20]. Another index type, proposed in [12], indexes, for each tuple and variable, the next tuple having a different value for the variable. Compression-based propagators compress the table in a form that allows a fast traversal. One of such compressed forms uses a trie for each variable [12]. Another example of compression-based techniques [7, 6] uses a *Multi Valued Decision Diagram* (MDD) to represent the table more efficiently. During propagation, the tries or MDD are traversed using the current domains to perform the pruning. These algorithms are constraint-based and have a time complexity of $O(r^2 \cdot d \cdot t)$ per table constraint along a branch in the search tree. This is a worst time complexity, corresponding to the case where there is few or no compression obtained with their respective encodings of the table. Compression and faster traversal can also be achieved by using compressed tuples, which represent a set of tuples [13, 27]. Propagators based on dynamic tables maintain the table by suppressing invalid tuples from it. The *or-tools* propagator [24] maintains such a dynamic table. It uses a bitset on the tuples of the table to maintain their validity. One bitset per literal (x, a) is also used for easy access of the tuples with value a for variable x. This propagator has a $O(r \cdot d \cdot t)$ time complexity per table constraint along a branch in the search tree. The STR algorithm [30] and its refined versions, STR2 and STR2+ [16], are also maintaining a dynamic table. They are constraint-based and scan only the previously valid tuples to extract the domain consistent values. The time complexity of STR2 and STR2+ is $O(r^2 \cdot d \cdot t)$ per table constraint along a branch in the search tree. This complexity is obtained by multiplying by $r \cdot d$ the complexity of one call to the propagator given in [16] while taking into account that the values of the domains can only be removed once along a branch in the search tree. The maximum number of calls to the propagator along one branch in the search tree is indeed $r \cdot d$. None of the previously presented propagators has the optimal $O(r \cdot t + r \cdot d)$ time complexity per table constraint. STR3 [19] has recently been introduced with an optimal time complexity per table constraint. Although the name might suggest it, STR3 is not an improvement of STR2. STR3 is a brand new GAC algorithm for table constraints. It is value-based (while STR2 and STR2+ are constraint-based), thus belonging to the last table constraint propagators category. STR3 starts by precomputing the set of initial supporting tuples for each literal (working with the indexes of those tuples). Those sets are not trailed during the search. Each valid literal has two special supports in its set: the last known valid tuple (called $curr$) and one valid support in this set (we call it $watched$). The tuple $curr$ is maintained during the search upon backtracking. Its property is that all the tuples after $curr$ are known to be invalid. The second special support is not backtracked during the search. This means that the two can be different. Upon the removal of a literal (x, a), a new valid tuple is searched for all the other literals having (x, a) in their $watched$ tuple. This search starts at $curr$ towards the head of the set. Once a new valid support is found, both $curr$ and $watched$ are updated for the literal being inspected. If none is found, then the literal is removed. STR3 can be seen

as a highly optimized version of GAC4, applying an idea similar to watched literals to the supports. A recent version of the MDD^c propagator, presented in [10] is both in the value based and compression based category. This propagator uses an MDD, as does MDD^c, but it never revisits parts of the MDD that do not need to be revisited. To achieve that, it switches from constraint based to value based. It also adds explanations to the MDD propagation, in an incremental fashion.

The AC5 Algorithm. AC5 [32, 8] is a generic value-based domain-consistency algorithm. In a constraint based approach, the propagation queue contains information about the constraints that need to re-enforce consistency. In a value-based approach, information on the removed values is also stored in the queue for the propagation. AC5 thus uses a queue Q of triplets (c, x, a) stating that the domain consistency of constraint c should be reconsidered because value a has been removed from $D(x)$. Specification 1 describes the main methods of AC5. In the postcondition of `enqueue`, Q_o represents the value of Q at call time. The propagators using AC5 should define their own `post` and `valRemove` methods. The generic AC5 algorithm, using those methods, is depicted in Algorithm 1. In all the pseudocodes presented in this paper, the assumed context is the resolution of a CSP $(X, D(X), C)$ and a propagation queue Q. The working principle of AC5 consists of two parts: initialization (`initAC5`) and queue propagation (`propagateQueueAC5`). In the initialization, the `post`$(c, \triangle)$ method is called once for each constraint c. Its role is to compute the inconsistent values of the constraint and initialize specific data structures required for the propagation. Each time a value is removed from a domain, `enqueue` puts the necessary information on the propagation queue. In the second phase of AC5, while there are triplets (c, y, b) on the queue, `valRemove`(c, y, b) is called so that the constraint c can reflect the removal of b from $D(y)$, possibly removing more literals. This dequeuing/enqueuing process is repeated until the queue becomes empty. At this point, the constraints using AC5 are domain consistent. As long as (c, x, a) is in the queue, it is algorithmically desirable to consider that value a is still in $D(x)$ from the perspective of constraint c. This is captured by the following definition.

Definition 1 The local view of a domain $D(x)$ wrt a queue Q for a constraint c is defined as $D(x, Q, c) = D(x) \cup \{a | (c, x, a) \in Q\}$.

For a constraint c, a queue Q and a set of variables $X = \{x_1 \dots x_n\}$, $D(X, Q, c)$ is defined as $\{D(x_1, Q, c), \dots, D(x_n, Q, c)\}$. For a table constraint c, a tuple σ is *Q-valid* if all its values belong to $D(\mathit{Vars}(c), Q, c)$. The central method of AC5 is the `valRemove` method, where the set $\triangle$ is the set of values becoming inconsistent because b is removed from $D(y)$. In this specification, b is a value that is no longer in $D(y)$ and `valRemove` computes the values (x, a) no longer supported in the constraint c because of the removal of b from $D(y)$. Note that values in the queue are still considered in the potential supports as their removal has not yet been reflected in this constraint. The minimal pruning $\triangle_1$ only deals with variables and values previously supported by (y, b). However, we give `valRemove` the possibility of achieving more pruning ($\triangle_2$), which is useful for table constraints.

```
enqueue(in x: Variable; in a: Value; in C1: Set of Constraints;
        inout Q: Queue)
// Pre: x ∈ X, a ∉ D(x), C1 ⊆ C
// Post: Q = Q0 ∪ {(c, x, a)|c ∈ C1, x ∈ Vars(c)}
post(in c: Constraint; out △: Set of Values)
// Pre: c ∈ C
// Post: △ = Inc(c) + initialization of specific data structures
valRemove(in c: Constraint; in y: Variable; in b: Value;
          out △: Set of Values)
// Pre: c ∈ C, b ∉ D(y, Q, c)
// Post: △1 ⊆ △ ⊆ △2 with △1 = Inc(c, D(X, Q, c)) ∩ Cons(c, y, b)
//                   and △2 = Inc(c)
```

Specification 1: The enqueue, post, and valRemove Methods for AC5

```
AC5(in X, C, inout D(X)){
// Pre: (X, D(X), C) is a CSP
// Post: D(X) ⊆ D(X)0, (X, D(X), C) equivalent to (X, D(X)0, C)
//       (X, D(X), C) is domain consistent
  initAC5(Q);
  propagateQueueAC5(Q);
}

initAC5(out Q){
  Q = ∅;
  C1 = ∅;
  forall(c in C){
    post(c, △);
    forall((x, a) in △){
      D(x) -= a;
      enqueue(x, a, C1, Q);
    }
    C1 += c;
  }
}

propagateQueueAC5(in Q){
  while Q != ∅ {
    select (c, y, b) in Q;
    Q = Q - (c, y, b);
    valRemove(c, y, b, △);
    forall((x, a) in △){
      D(x) -= a;
      enqueue(x, a, C\{c}, Q);
    }
  }
}
```

Algorithm 1: The AC5 Algorithm.

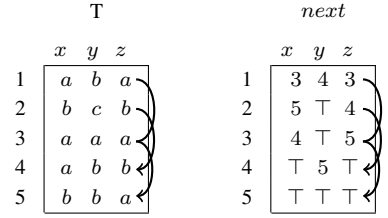


Fig. 1: Example of a $next$ data structure of a table T (arrow pointers for variable z only).

3 Efficient Value-Based Algorithms for Table Constraints

Our algorithms use a data structure *FS* memorizing first supports. Intuitively *FS*$[x, a]$ is the index of the first Q-valid support of the variable value pair (x, a). It is thus equivalent to the *last* structure used in GAC2001 algorithms. To speed up the table traversal, our algorithms use a second data structure called $next$ that links all the elements of the table sharing the same value for a given variable. More formally, for a given table constraint c, *FS* and *next* satisfy the following invariant (called FS-invariant) before dequeuing an element from Q.

$$\begin{array}{l}\forall x \in \mathit{Vars}(c)\ \forall a \in D(x, Q, c) : FS[x, a] = i \Leftrightarrow \\ \quad \sigma_{c,i}[x] = a \wedge i \neq \top \wedge \sigma_{c,i} \in D(\mathit{Vars}(c), Q, c) \\ \quad \wedge\ \forall j < i : \sigma_{c,j}[x] = a \Rightarrow \sigma_{c,j} \notin D(\mathit{Vars}(c), Q, c) \\ \forall x \in \mathit{Vars}(c)\ \forall 1 \leq i \leq c.length : next[x, i] = Min\{j | i < j \wedge \sigma_{c,j}[x] = \sigma_{c,i}[x]\}\end{array}$$

The $next$ data structure, illustrated in Figure 1, is static as it does not depend on the domain of the variables. However, *FS* must be trailed during the search.

Methods `postTC` and `valRemoveTC` are given in Algorithms 2 and 3. TC is included in the methods and algorithms names to underline the fact that those are propagators for Table Constraints. Note that the $\triangle$ computed by `valRemoveTC` corresponds to $\triangle_1$ in Specification 1. Method `valRemoveTC` uses the `seekNextSupportTC` method (Algorithm 4) which searches the next Q-valid tuple of a literal. Abstract method `isQValidTC(c,i)` tests whether $\sigma_{c,i}$ is Q-valid (i.e., $\sigma_{c,i} \in D(\mathit{Vars}(c), Q, c)$) and can be implemented in many ways. One simple way is to record the Q-validity of tuples in some data structure, initialized in method `initSpecStructTC` and updated in method `setQInvalidTC`. Method `postTC` initializes the FS and $next$ data structures and returns the set of inconsistent values. Method `valRemoveTC` has only to consider the tuples in the $next$ chain starting at $FS[y, b]$. The tuples before are invalid and cannot be the first support of any other Q-valid literal. When one of the traversed tuples $\sigma_{c,i}$ is the first support of an element $a = \sigma_{c,i}[x]$, a new support $FS[x, a]$ must be found. Indeed, $\sigma_{c,i}$ is no more Q-valid. If such a new support does not exist, then (x, a) belongs to the set $\triangle$. Method `valRemoveTC` thus computes the set $\triangle$ and maintains the FS-invariant. The AC5 algorithm with the `postTC` and `valRemoveTC` implementation for table constraint is called AC5TC (AC5 for Table Constraints).

Proposition 1 *Assuming that `initSpecStructTC` and `setQInvalidTC` have a time complexity of $O(r \cdot t + r \cdot d)$ and $O(1)$ respectively and allow a correct implementation of `isQValidTC` to have a complexity of $O(r)$, then AC5TC is correct and has a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint along a branch in the search tree.*

Proof Assuming a correct implementation of `isQValidTC`, AC5TC is correct. Indeed, `postTC` and `valRemoveTC` respect their specification (Specification 1). Not considering `initSpecStructTC`, method `postTC` has a time complexity of $O(r \cdot t + r \cdot d)$. After the `postTC` method, the domain size of x is $O(t)$ since each value in $D(x)$ has at least one support in the table. We now establish the complexity of all executions of `valRemoveTC` for a given table constraint, assuming this table constraint is one of the constraints of the CSP on which domain consistency is achieved. Consider first all executions of `valRemoveTC` without line 12. For a given variable y, these executions follow the different $next$ chains of the variable y. The chains for all values of y have a total number of t elements. The

```
1  postTC(in c: Constraint;out △: Set of Values){
2  // Pre: c ∈ C, c is a table constraint
3  // Post: △ = Inc(c) + initialization of the next, FS and specific data structures
4      △ = ∅;
5      initSpecStructTC(c);
6      forall(x in Vars(c), a in D(x)) c.FS[x,a]=⊤;
7      forall(x in Vars(c), i in 1..c.length)  c.next[x,i] = ⊤;
8      forall(i in c.length..1)
9         if (σ_{c,i} in D(Vars(c))){
10           forall(x in Vars(c)){
11             c.next[x,i] = FS[x,σ_{c,i}[x]];
12             c.FS[x,σ_{c,i}[x]] = i;
13           }
14        }
15        else setQInvalidTC(c,i);
16     forall(x in Vars(c),a in D(x))
17        if(c.FS[x,a]==⊤) △ += (x,a);
18 }
```

Algorithm 2: Method `postTC` for Table Constraints

```
1  valRemoveTC(in c: Constraint;in y: Variable; in b: Value;
2              out △: Set of Values) {
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y, Q, c)
4  // Post: △ = Inc(c, D(X, Q, c)) ∩ Cons(c, y, b)
5      △ = ∅;
6      i = c.FS[y,b];
7      while(i!=⊤){
8         setQInvalidTC(c,i);
9         forall(x in Vars(c): x!=y){
10           a = σ_{c,i}[x];
11           if (c.FS[x,a]==i){
12              c.FS[x,a] = seekNextSupportTC(c,x,i);
13              if(c.FS[x,a]==⊤ && a in D(x))  △ += (x,a);
14           }
15        }
16        i = c.next[y,i];
17     }
18 }
```

Algorithm 3: Method `valRemoveTC` for Table Constraints.

```
1  function seekNextSupportTC(in c: Constraint; in x: Variable;
2                             in i: Index)  : Index {
3  // Pre: c ∈ C, c is a table constraint, x ∈ Vars(c), 1 ≤ i ≤ c.length
4  // Post: return the first index j greater than i of a Q−valid tuple with σ_{c,j}[x] == σ_{c,i}[x]
5      i = c.next[x,i];
6      while(i!=⊤){
7         if(isQValidTC(c,i))  return i;
8         i = c.next[x,i];
9      }
10     return ⊤;
11 }
```

Algorithm 4: Function `seekNextSupportTC` for Table Constraints.

complexity of lines 9–16 (without line 12) is $O(r)$. Since the table has r variables, the complexity of all `valRemoveTC` executions during the fixed point (without line 12) is thus $O(r^2 \cdot t)$, assuming a $O(1)$ complexity of `setQInvalidTC`. Consider now all executions of line 12 in `valRemoveTC` for a variable x. Since line 12 always increases the value of $FS[x, a]$ in the $next$ chain of (x, a), we have a global complexity of $O(V \cdot t)$ for the variable x, where V is the time complexity of `isQValidTC`. All executions of line 12 in `valRemoveTC` thus take time $O(V \cdot r \cdot t)$. The time complexity of all executions of `valRemoveTC` is then $O(r^2 \cdot t + V \cdot r \cdot t)$. With a $O(r)$ `isQvalidTC`, this complexity is $O(r^2 \cdot t)$, giving AC5TC a complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint along a branch in the search tree. ■

Even with a $O(1)$ the time complexity of `isQValidTC`, the algorithm has a complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint along a branch in the search tree. It is thus not optimal but its implementations turn out, in the experiments, to be more efficient than state-of-the-art algorithms on some classes of problems.

```
1  initSpecStructTC-Bool(in c: Constraint) {
2     forall(i in 1..c.length)  c.isQValid[i] = true;
3  }
4  function isQValidTC-Bool(in c: Constraint;in i: Index) {
5  // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
6  // Post: returns σ_{c,i} ∈ D(X, Q, c)
7     return c.isQValid[i];
8  }
9  setQInvalidTC-Bool(in c: Constraint;in i: Index) {
10 // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
11    c.isQValid[i] = false;
12 }
```

Algorithm 5: Implementation of the specific methods of AC5TC-Bool

```
1  initSpecStructTC-Sparse(in c: Constraint){
2     forall(i in 1..c.length){
3        c.Map[i] = i;
4        c.Dyn[i] = i;
5     }
6     c.size = c.length;
7  }
8  function isQValidTC-Sparse(in c: Constraint; in i: Index;out b: Bool){
9  // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
10 // Post: return (σ_{c,i} ∈ D(X, Q, c))
11    return (c.Map[i] <= c.size);
12 }
13 setQInvalidTC-Sparse(in c: Constraint; in i: Index){
14 // Pre: c ∈ C, c is a table constraint and 1 ≤ i ≤ c.length
15    c.Dyn[c.Map[i]] = c.Dyn[c.size];
16    c.Dyn[c.size] = i;
17    c.Map[c.Dyn[c.Map[i]]] = c.Map[i];
18    c.Map[i] = c.size;
19    c.size--;
20 }
```

Algorithm 6: Implementation of the specific methods of AC5TC-Sparse

We now present two implementations of AC5TC. They differ in the implementations of methods `isQValidTC`, `setQInvalidTC` and `initSpecStructTC`. AC5TC-Bool, the first implementation of AC5TC, is shown in Algorithm 5. It uses a data structure *isQValid*$[i]$ to record the Q-validity of the element $\sigma_{c,i}$. It satisfies invariant *isQValid*$[i] \Leftrightarrow \sigma_{c,i} \in D(\mathit{Vars}(c), Q, c)$ before dequeuing an element from Q ($1 \leq i \leq c.\mathit{length}$). The data structure must be trailed as it depends on the domains. The methods for Q-validity are given in Algorithm 5. As the methods `isQValidTC-Bool` is correct, AC5TC-Bool is correct. The time complexity of `isQValidTC-Bool` and `setQInvalidTC-Bool` is $O(1)$ and `initSpecStructTC-Bool` is $O(t)$. The time complexity of AC5TC-Bool is then $O(r^2 \cdot t + r \cdot d)$ per table constraint.

AC5TC-Bool must trail the *isQValid* boolean array. We now propose an implementation that only trails one integer, building upon an idea in STR, STR2 and STR2+ [30, 16] originally described in [5]. This structure, called Sparse Set, keeps elements that have been removed at the end of the table, with a single variable size representing the boundary between present (before position size) and removed elements (after position size). When an element is removed, it is swapped with the element at position size and size is decremented by one. This representation uses two arrays Map and Dyn to represent the sparse set. Map contains, for each entry of the set, its position in the array Dyn. Dyn contains the elements of the set, before the index size and the ones which have been removed, after size. Figure 2 depicts the structures representing the set $\{1, 2, 3, 4, 6, 7, 9, 10\}$ from which 5 and 8 have previously been removed. Here, as it will be the case in our propagator, we have that the values are successive integer values between 1 and n. This is not a necessary condition. Figure 3 illustrates the remove operation. The size variable must be trailed but the arrays Map and *Dyn* do not need to. When restoring size, the elements of the set are automatically restored, albeit at a different position in the table. This is sometimes called semantic backtracking [31]. For this implementation, the Sparse Sets are used to keep track of the set of Q-valid tuple indexes in the table of the constraint. Figure 4 illustrates the use of the sparse sets for this implementation. Instead of trailing t booleans as AC5TC-Bool is doing, AC5TC-Sparse only has to trail size (one integer). More formally, for a given table constraint c, the data structures satisfy the following invariants before dequeuing an element from Q:

$$\forall\, 1 \leq i \leq c.\mathit{length} : (\mathit{Map}[i] \leq \mathit{size} \Leftrightarrow \sigma_i \in D(\mathit{Vars}(c), Q, c)\ ) \wedge\ \mathit{Dyn}[\mathit{Map}[i]] = i$$

Testing the Q-validity of a tuple index i in such set is easy: it suffices to test whether $\mathit{Map}[i]$ is less or equal than size. This is a $O(1)$ operation.

The implementations are given in Algorithm 6 and the algorithm is called AC5TC-Sparse. The time complexity of `isQValidTC-Sparse` and `setQInvalidTC-Sparse` is $O(1)$ and `initSpecStructTC-Sparse` takes time $O(t)$. The time complexity of AC5TC-Sparse is thus $O(r^2 \cdot t + r \cdot d)$.

The space complexities of both AC5TC-Bool and AC5TC-Sparse are $\Theta(r \cdot t + r \cdot d)$. Indeed, the next structure is $\Theta(r \cdot t)$ since each tuple appears in r chains, FS is $\Theta(r \cdot d)$ and both $\mathit{isQValid}$ and $\mathit{Map}/\mathit{Dyn}$ are $\Theta(t)$. AC5TC-Bool has $\Theta(r \cdot d)$ integers and $\Theta(t)$ boolean to trail during the search: it must trail FS and $\mathit{isQValid}$. AC5TC-Sparse only has to trail FS and one integer, so $\Theta(r \cdot d)$ integers. The cost the algorithms have to pay during backtracking is proportional to the number of changes in the structures. The nature, integer or boolean, of the elements of the structure doesn't matter: a cost of $O(1)$ has to be paid to restore them. Each change is recorded in the trailing record of the algorithms. We thus compare here the number of modifications to their structures, which is the size of their trailing record. For

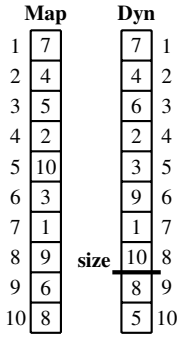
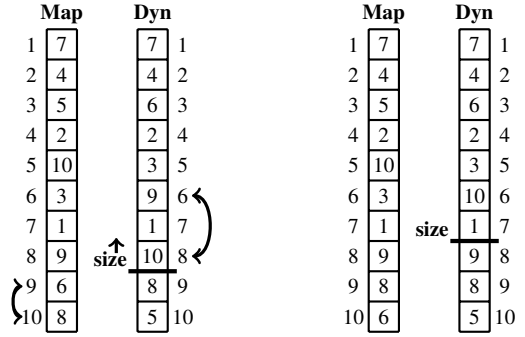


Fig. 2: The two arrays representing the Sparse Set $\{1, 2, 3, 4, 6, 7, 9, 10\}$



(a) Operations to remove 9 from the set

(b) Resulting Dyn and Map

Fig. 3: Removing 9 from the set $\{1, 2, 3, 4, 6, 7, 9, 10\}$ (5 and 8 previously removed)

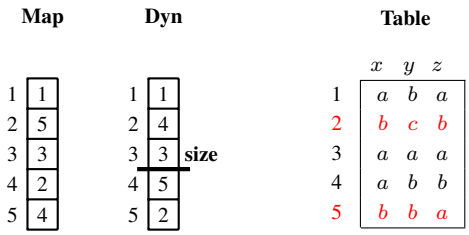


Fig. 4: AC5TC-Sparse using Sparse sets to keep track of Q-valid tuples. In this example, tuples 2 and 5 are Q-invalid.

one table constraint, along a branch in the search tree containing m nodes, if k tuples of the table are found Q-invalid, the size of the trailing record due to FS is $O(r \cdot k)$. In addition, AC5TC-Bool has k elements in its trailing record due to $isQValid$ and AC5TC-Sparse has at most m elements in its record, because only $size$ is trailed. If at most one tuple is found Q-invalid in each node, the records have the same size but usually, we have $m < k$. Also, the contribution to the record size due to the Q-validity structure seems small compared to the contribution of FS. However the bound on the number of modifications to FS corresponds

to a rare case. Indeed, it will only be attained if each removed tuple updates the first support of $r - 1$ variables. In practice, the choice of the Q-validity structure has a large impact on the performances of the propagators.

4 A Variation Based on Recomputation

Using the Q-validity of the tuples while traversing the $next$ structure has a drawback. A literal that is no longer domain consistent may have its first support updated multiple times before being removed. In a worst case scenario, a literal with a lot of Q-valid (but not valid) supports may have its first support updated to each of the Q-valid supports before being removed. Example 1 illustrates this problem.

Example 1 In a binary table constraint c where $Vars(c) = \{x, y\}$, let's suppose a literal (y, b) has only 3 supports (ordered as they are in the table): σ_1, σ_2 and σ_3. Let's suppose that $\sigma_1[x] = a$, $\sigma_2[x] = b$ and $\sigma_3[x] = c$. If the propagation queue contains (x, a), (x, b) and (x, c) and they are popped in that order, the first support for (y, b) (initially σ_1) will be updated to σ_2, then to σ_3 and finally to $\top$.

The solution to this problem is to work directly with the validity of the tuples. In the previous scenario, the literal would have been removed the first time a new valid support would have been searched. We thus propose a variation of the AC5TC algorithm, called AC5TC-Recomp, that works with the validity of the tuples. AC5TC-Recomp does not require any data structure to store Q-validity information. Rather, the validity information is recomputed as needed and not stored. Even with this switch from Q-validity to validity, AC5TC-Recomp still uses the same $next$ structure as used by AC5TC. However, the *FS* structure has to be slightly changed together with its invariant. The new structure is called *FUS* and stores, for each literal, the index of the first useful support in the table. Its new invariant can be found below and is satisfied before dequeuing an element of Q.

$$\begin{aligned}
&\forall x \in Vars(c), \forall a \in D(x) : \\
&\quad FUS[x, a] = i \Leftrightarrow [\sigma_{c,i}[x] = a \wedge i \neq \top \wedge \sigma_{c,i} \in D(Vars(c), Q, c) \\
&\qquad\qquad \wedge (\forall j < i : \sigma_{c,j}[x] = a \Rightarrow \sigma_{c,j} \notin D(Vars(c)))]
\end{aligned}$$

$$\begin{aligned}
&\forall x, y \in Vars(c), \forall a \in D(x), b \in D(y, Q, c) \backslash D(y) : \\
&\qquad (FUS[x, a] = i \wedge \sigma_{c,i}[y] = b) \Rightarrow FUS[y, b] \leq FUS[x, a]
\end{aligned}$$

The first part of the invariant states that, for a literal in the domain, the first useful support is Q-valid and different from $\top$. It also states that all the candidate supports before the first useful support are invalid. This is the first difference with the invariant for FS in AC5TC. Recall that, in the FS invariant, the preceding candidate supports were Q-invalid. This difference arises because AC5TC-Recomp is not using Q-validity information. When updating the first useful support of a tuple, it may skip Q-valid candidate supports that are invalid. The second part of the invariant is dedicated to ensure that the update process will not miss FUS update for some literal. This will be explained in more details later, together with the update process.

The `post` method of AC5TC-Recomp is very similar to the `postTC` method of Algorithm 2. It also initializes the $next$ data structure and *FUS* is initialized the same way *FS* is in

postTC. No Q-validity is used in either of the post methods and *FS* and *FUS* are exactly the same after the initialization. The only difference between postTC and the post method of AC5TC-Recomp is that AC5TC-Recomp does not perform calls to initSpecStructTC nor to setQInvalidTC.

The valRemoveTC-Recomp method is given in Algorithm 7. This method is similar to valRemoveTC (Algorithm 3). Indeed, when called for a constraint c, a variable y and a value b, valRemoveTC-Recomp cycles through the tuples $\sigma_{c,i}$ where $\sigma_{c,i}[y] = b$. It starts at $FUS[y, b]$ because the tuples before $FUS[y, b]$ are invalid and cannot be the first useful support for any valid literal. This is granted by the FUS invariant. For each of those tuples $\sigma_{c,i}$, a new support is searched for each literal (x, a) for which $FUS[x, a] = i$. If no new support is found, valRemoveTC-Recomp includes (x, a) in $\triangle$, as it is done in valRemoveTC. However, valRemoveTC-Recomp and valRemoveTC have significant differences. The first one is the use of seekNextSupportTC-Recomp method to update *FUS*. This method searches for the next valid support, traversing the $next$ chain. Recall that seekNextSupportTC (Algorithm 4) updates *FS* searching for the next Q-valid support. Another difference is the test $\sigma_{c,i}[x] \in D(x)$ of line 8. This test is the equivalent of the one on line 13 in Algorithm 3. This test is needed here on line 8 to maintain the second part of the FUS invariant. This part of the invariant guarantees that, for each literal in the queue, its first useful support is before the first useful support of the valid literals in this tuple. This allows valRemoveTC-Recomp(c, y, b) to look for updates to make from $FUS[y, b]$ towards the end of the table. In order to see the need for the second part of the invariant, let's suppose it is not present and the test $\sigma_{c,i}[x] \in D(x)$ is pushed inside line 12 (as it is in Algorithm 3). Let's suppose we are executing valRemoveTC-Recomp for the literal (y, b). Let's also suppose that the first useful support of a literal (x, a) in the propagation queue contains (y, b). Without the test $\sigma_{c,i}[x] \in D(x)$ of line 8, a new valid support would be searched for (x, a) to update $FUS[x, a]$. This would automatically set $FUS[x, a]$ to $\top$ because all supports of (x, a) contain (x, a) and are thus invalid. Setting $FUS[x, a]$ to $\top$ would prevent the call to valRemoveTC-Recomp for (x, a) to update FUS for the literals that have a tuple with (x, a) as first useful support. This may lead the algorithm to leave non-GAC literals out of $\triangle$. This characteristic is already implicitly present in the FS invariant (section 3), thanks to the Q-validity of the first support granted for the valid literals and the literals in the queue and the Q-invalidity of the candidate supports before. This guarantees that the first support of each literal in the queue is before the first support of the literals in this tuple.

From a complexity point of view, it may seem inefficient to test the validity of the tuples when searching for a new support. The test $\sigma_{c,i} \in D(Vars(c))$ is in $O(r)$. Using the validity has however an advantage. It allows valRemoveTC to output a $\triangle$ set larger than the previous algorithms. The validity information is stronger than the Q-validity information since each valid tuple is also Q-valid but not the contrary. With this larger $\triangle$, the domains at the fixed point will be the same for all algorithms (since all compute domain consistency) but AC5TC-Recomp might need fewer calls to valRemoveTC-Recomp to achieve this fixed point.

Proposition 2 *AC5TC-Recomp is correct. It has a time complexity of $O(r^2 \cdot t + r \cdot d)$ and a space complexity of $O(r \cdot t + r \cdot d)$ per table constraint along a branch in the search tree.*

Proof Both methods postTC-Recomp and valRemoveTC-Recomp satisfy their specifications (Specification 1). AC5TC-Recomp is hence correct. The post method of AC5TC-

```
1  valRemoveTC-Recomp(in c: Constraint;in y: Variable; in b: Value;
2            out △: Set of Values) {
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y, Q, c)
4  // Post: Inc(c, D(X, Q, c)) ∩ Cons(c, y, b) ⊆ △ ⊆ Inc(c) ∩ Cons(c, y, b)
5      △ = ∅;
6      i = c.FUS[y, b];
7      while(i!=⊤) {
8          forall(x in Vars(c): x!=y && σ_{c,i}[x] ∈ D(x)) {
9              a = σ_{c,i}[x];
10             if (c.FUS[x, a]==i) {
11                 c.FUS[x, a] = seekNextSupportTC-Recomp(c, x, i);
12                 if(c.FUS[x, a]==⊤)  △ += (x, a);
13             }
14         }
15         i = c.next[y, i];
16     }
17 }
```

Algorithm 7: Method `valRemoveTC-Recomp` for Table Constraints.

```
1  function seekNextSupportTC-Recomp(in c: Constraint; in x: Variable;
2                                    in i: Index)  : Index {
3  // Pre: c ∈ C, c is a table constraint, x ∈ Vars(c), 1 ≤ i ≤ c.length
4  // Post: return the first index j greater than i of a valid tuple with σ_{c,j}[x] == σ_{c,i}[x]
5      i = c.next[x, i];
6      while(i!=⊤) {
7          if(σ_{c,i} ∈ D(Vars(c)))  return i;
8          i = c.next[x, i];
9      }
10     return ⊤;
11 }
```

Algorithm 8: The `seekNextSupportTC-Recomp` Function of AC5TC-Recomp.

Recomp has the same complexity as the one of AC5TC: $O(r \cdot t + r \cdot d)$. The complexity of all the executions of `valRemoveTC-Recomp` is $O(r^2 \cdot t)$ per table constraint. The justification of the complexity is similar to the one for `valRemoveTC` (Proposition 1). For a variable y, all the executions of `valRemoveTC-Recomp` follow the chains in the $next$ structure for each different value of the variable. The total number of elements in those chains is t since the tuples have one value per variable. With r variables, the lines 8 to 15 are executed $O(r \cdot t)$ times. Without line 11, those lines have a complexity of $O(r)$. Without line 11, the complexity of all the executions of `valRemoveTC-Recomp` is $O(r^2 \cdot t)$. For a particular variable, the loop of line 6 in `seekNextSupportTC-Recomp` can only be executed $O(t)$ times for all its values during the entire fixed point. Indeed, the next chains are traversed at most once. The test $\sigma_{c,i} \in D(Vars(c))$ of line 7 in `seekNextSupportTC-Recomp` being $O(r)$, all the executions of line 11 of `valRemoveTC-Recomp` are $O(r^2 \cdot t)$. Hence the complexity of $O(r^2 \cdot t)$ for `valRemoveTC-Recomp` during the fixed point. ■

The space complexity of AC5TC-Recomp is $\Theta(r \cdot t + r \cdot d)$ as it uses the same $next$ structure as used in the previous Section and FUS is $\Theta(r \cdot d)$. AC5TC-Recomp has to trail FUS, which means trailing $\Theta(r \cdot d)$ integers. When the fixed point is reached, the FUS structure is the same as the FS structure in the previous algorithms. The number of modifications to the structure AC5TC-Recomp has to store in its trailing record for FUS is thus the same as the number of modifications the previous algorithms have to store for FS. In addition, along

a branch in the search tree containing m nodes, if k tuples are found Q-invalid, AC5TC-Bool has k elements in its trailing record while AC5TC-Sparse has at most only m integers in its record. AC5TC-Recomp doesn't have to store those modifications in its record.

Despite being non optimal, AC5TC-Recomp improves, in the experiments, state-of-the-art algorithms on some classes of problems. It is the (previously unpublished) table constraint algorithm of the Comet system.

5 Optimal Algorithms for Table Constraints

In all of the previously presented algorithms, the successive updates of their respective support structure may revisit the same tuples multiple times. This comes from the static characteristic of the $next$ structure. Indeed, there is no guarantee that a tuple visited in a particular $next$ chain, and found to be Q-invalid or invalid, will not be revisited later in another chain. This redundant work has a cost. This cost is particularly penalizing for AC5TC (Section 3). In method `valRemoveTC` (Algorithm 3), all the executions of the `seekNextSupportTC` (line 12 of Algorithm 3) take $O(r \cdot t)$ assuming `isQValidTC` takes constant time. However, the total time complexity of all the executions of `valRemoveTC` is $O(r^2 \cdot t)$. This makes AC5TC reach the non optimal $O(r^2 \cdot t + r \cdot d)$ time complexity per table constraint. Recall that the optimal time complexity per table constraint is $O(r \cdot t + r \cdot d)$. To remedy this situation, the idea is to use a dynamic collection of the supports for each literal instead of the static $next$ structure. Those dynamic structures thus keep the Q-valid supports for the values in $D(X, Q, c)$. This allows to avoid revisiting tuples because as soon as a tuple is detected Q-invalid, it is removed from the active collections it belongs to. Those collections are stored in a single structure, called Col. More formally, for a table constraint c, the structure containing the collections, Col, satisfies the following invariant before dequeuing an element of Q:

$$\forall x \in Vars(c), \forall a \in D(x, Q, c) :$$
$$Col[x, a] = \{1 \leq i \leq c.length \mid \sigma_{c,i}[x] = a \wedge \sigma_{c,i} \in D(Vars(c), Q, c)\}$$

This invariant simply states that for each variable x and each value in its extended domain $D(x, Q, c)$, its collection contains all its Q-valid supports. In order to allow different implementations for this collection structure, we have designed a generic propagator. This propagator uses the abstract methods specified in Specification 2. Those methods simply form iterators over Col. They enclose all interactions with the collections in Col. For the iterator requirement, we assume that no remove operation is performed on a collection when iterating over it. We will refer to them as the specific functions as they are specific to the concrete propagators implementing the generic propagator.

The optimal generic algorithm for table constraints is called AC5TCOpt. The `post` and `valRemove` methods for AC5TCOpt are presented respectively in Algorithms 9 and 10. Method `postTCOpt` first computes the initial collection Col_{init}. Col_{init} is represented by a matrix containing one array per literal. For a literal, its array in Col_{init} contains the indexes of its supporting tuples. `postTCOpt` then initializes the specific structures of the propagator by calling `initSpecStruct`. It then removes all the values with no valid support. Method `valRemoveTCOpt`(c, y, b) uses methods `firstInCollection`(c, y, b) (line 6), `nextInCollection`(c, y, i) (line 15) and the test $i \neq \top$ (line 7) to traverse the

```
1  initSpecStruct(in c: Constraint, in Col_init: Matrix of Arrays)
2  //pre: the collections in Col_init are sorted
3  //initializes specific structures implementing Col to the
4  //initial collections contained in the matrix Col_init
5  //in Col_init, a collection is represented by an array.
6
7  function firstInCollection(in c: Constraint; in x: Variable;
8      in a: Value): Index
9  //returns the first element in Col[x, a]
10
11 function nextInCollection(in c: Constraint; in x: Variable;
12     in i: Index): Index
13 //returns the element following i in Col[x,a] where a = σ_{c,i}[x]
14
15 //functions firstInCollection, nextInCollection and the test
16 //nextInCollection≠ ⊤ form iterators over the collections
17 //in Col
18
19 function isCollectionEmpty(in c: Constraint; in x: Variable;
20     in a: Value): Boolean
21 //returns true iff Col[x,a] is empty
22
23 removeFromCollection(in c: Constraint,in x: Variable,in i: Index)
24 //removes i from Col[x,a] where a = σ_{c,i}[x]
```

Specification 2: The abstract methods used by the generic optimal table constraint propagator

collection for (y, b). Those are the only tuples that become Q-invalid when (y, b) is popped out of Q. Those tuples are removed from the collections they belong to and the literals left without any Q-valid support are included in $\triangle$.

```
1  postTCOpt(in c: Constraint; out △: Set of Values){
2  // Pre: c ∈ C, c is a table constraint
3  // Post: △ = Inc(c) + initialization of the data structures
4      △ = ∅;
5      forall(x in Vars(c),a in D(x)) Col_init[x, a]=[];
6      forall(i in 1..c.length: σ_{c,i} in D(Vars(c)))
7          forall(x in Vars(c))
8              Col_init[x, σ_{c,i}[x]].append(i);
9      initSpecStruct(c,Col_init);
10     forall(x in Vars(c),a in D(x))
11         if(isCollectionEmpty(c, x, a)) △ += (x,a);
12 }
```

Algorithm 9: The post method of optimal AC5TCOpt table constraint propagator

Proposition 3 *Assuming that* `initSpecStruct`, `firstInCollection`, `nextInCollection`, `isCollectionEmpty` *and* `removeFromCollection` *are correct, AC5TCOpt is correct. Assuming a complexity of* $O(r \cdot t + r \cdot d)$ *for* `initSpecStruct`*, and* $O(1)$ *for the other specific functions, the time complexity of AC5TCOpt is the optimal* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

```
1  valRemoveTCOpt(in c: Constraint; in y: Variable; in b: Value;
2                 out △: Set of Values) {
3  // Pre: c ∈ C, c is a table constraint and b ∉ D(y, Q, c)
4  // Post: △ = Inc(c, D(X, Q, c)) ∩ Cons(c, y, b)
5    △ = ∅;
6    i = firstInCollection(c, y, b);
7    while(i ≠ ⊤){
8      forall(x in Vars(c) : x ≠ y){
9          removeFromCollection(c, x, i);
10         a = σ_{c,i}[x];
11         if(isCollectionEmpty(c, x, a) && a in D(x)){
12             △ += (x, a);
13         }
14     }
15     i = nextInCollection(c, y, i);
16   }
17 }
```

Algorithm 10: The valRemove method of optimal AC5TCOpt table constraint propagator

Proof After the execution of Method `postTCOpt`, the Col invariant is respected. Let's suppose that the Col invariant is true before dequeuing a literal (y, b) from Q. As the invariant is verified before the execution, the only tuples that need attention are the tuples becoming Q-invalid. Since the specific methods are correct, `valRemoveTCOpt` iterates through all the previously Q-valid tuples $\sigma_{c,i}$ where $\sigma_{c,i}[y] = b$. Those tuple indexes are removed from all the sets they belong to. The Col invariant is thus restored. With the Col invariant, it is straightforward to prove that AC5TCOpt is correct. Indeed, both `postTCOpt` and `valRemoveTCOpt` meet Specification 1 since the $\triangle$s are filled with the literals for which the collection in Col is empty. With a $O(r \cdot t + r \cdot d)$ complexity for `initSpecStruct` and a $O(1)$ complexity for `isCollectionEmpty`, postTCOpt is obviously $O(r \cdot t + r \cdot d)$. The complexity of `valremoveTCOpt` is $O(r \cdot t)$. Indeed, each execution of lines 6 to 16 leads to different values for i in $\{1, \ldots, t\}$. This is granted by the fact that the indexes of the visited tuples are removed from all the collections they belong to. The lines 8 to 15 are thus executed at most t times along a branch in the search tree. The complexity of those lines being $O(r)$, the overall complexity of `valRemove` is $O(r \cdot t)$ per table constraint (assuming the presence of other constraints on which domain consistency is also applied). ■

We now present two different implementations of AC5TCOpt. They differ in the data structure they use to backup the implementation of the functions `firstInCollection`, `nextInCollection`, `isCollectionEmpty` and `removeFromCollection`.

5.1 AC5TCOpt-Tr

The first propagator is reusing an idea similar to the $next$ structure of AC5TC (Section 3) and AC5TC-Recomp (Section 4). In order to implement the specific functions of AC5TCOpt, iterating through the collections in Col, this new structure has to be dynamic and it has to allow removal of tuples in the collection in $O(1)$. Indeed, the collections in Col depend on the domains of the variables and thus, have to change during the search. This is why we used two arrays to represent the index collections in Col: $nextTr$ and $predTr$. The first one,

$nextTr$, contains, for an index i and a variable x, the next tuple index in $Col[x, \sigma_{c,i}[x]]$. The second one, $predTr$, contains the preceding tuple index in $Col[x, \sigma_{c,i}[x]]$. This propagator also uses an array, called FS, referring, for a literal (x, a) to the first tuple index in $Col[x, a]$. The name FS has been chosen for the similarities with the FS array in AC5TC (Section 3). The *nextTr*, *predTr* and FS data structures should be trailed as Col depends on the current domains. For a variable x and a value $a \in D(x, Q, c)$, $Col[x, a]$ can be enumerated by following the $nextTr$ chain, starting at $FS[x, a]$. The order between the tuples in $nextTr$ and $predTr$ is fixed to the table order. The collections in Col are thus here ordered with respect to this order. As the $nextTr$ and $predTr$ are trailed through the execution of the propagator, we call the AC5Opt algorithm using those structures AC5TCOpt-Tr (for AC5 Optimal Table Constraint Propagator with Trailing).

The implementation of the specific functions (Specification 2) of AC5TCOpt-Tr is presented in Algorithm 11. Method `initSpecStruct` initializes $nextTr$, $predTr$ and FS. As explained before, FS contains the beginning of the collection for each different literal, in this case, the smallest index (the arrays in Col_{init} are sorted). The functions `firstInCollection` and `nextInCollection` are straightforward. For `isCollectionEmpty`, if the smallest index of an element in a collection is $\top$, that means that the collection for the literal is empty. The `removeFromCollection` also takes into account the increasing order of the indexes in the structures. It has to distinguish between 3 cases. In the first one, the index to remove is the first one (line 34). In that case, the first one is set to be the index following it. This effectively removes index i from the collection because the traversal of the collection is performed from FS to the end of the table. The second case (line 36) is the one where the index to remove is after the first one. In this case, it must be removed from the $nextTr$ and $predTr$ structures to remove it from the collection. The third case is the case where i is smaller than $FS[x, a]$. In this case, nothing has to be done: thanks to the traversal order, i will never be visited. It is thus not in the collection.

Proposition 4 *The implementation of* `initSpecStruct`*,* `firstInCollection`*,* `nextInCollection`*,* `isCollectionEmpty` *and* `removeFromCollection` *is correct. The time complexity of* `initSpecStruct` *is* $O(r \cdot t + r \cdot d)$*. All the other methods are* $O(1)$*. The time complexity of AC5TCOpt-Tr is the optimal* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof The proof that the different functions respect their individual specification (Specification 2) is straightforward. The last part of the specification to prove is the requirement that the functions `firstInCollection`, `nextInCollection` and the test `nextInCollection`$\neq \top$ form iterators over the collections in Col. It is is granted by the fact that FS is alway referring to the first element in its collection and that the only elements removed from the $nextTr$ and $predTr$ chains are the ones removed from the collections with `removeFromCollection`. The complexity of `initSpecStruct` is $O(r \cdot t + r \cdot d)$ because the total number of supports for a variable is $O(t)$, hence is the number of elements in the Col structures for a variable. This guarantees that the loop in line 10 is $O(t)$ for all the values of a variable. The loop in line 4 is thus $O(r \cdot t + r \cdot d)$. AC5TCOpt-Tr is then $O(r \cdot t + r \cdot d)$ ∎

AC5TCOpt-Tr has a space complexity of $\Theta(r \cdot t + r \cdot d)$ since $nextTr/predTr$ are $\Theta(r \cdot t)$ and FS is $\Theta(r \cdot d)$. All those structures have to be trailed during the search. AC5TC-Tr has thus $\Theta(r \cdot t + r \cdot d)$ integers to backtrack. Compared to the previous algorithms, the difference

```
1  initSpecStruct(in c: Constraint, in Col_init: Matrix of Arrays){
2     forall(x in Vars(c), i in 1..c.length)
3        c.nextTr[x,i] = ⊤; c.predTr[x,i] = ⊥;
4     forall(x in Vars(c), a in D(x)){
5        nSup=len(Col_init[x,a]);
6        if(nSup==0)
7           c.FS[x,a]=⊤;
8        else
9           c.FS[x,a]=Col_init[x,a][1];
10          forall(j in 1..nSup-1){
11             c.nextTr[x,Col_init[x,a][j]]=Col_init[x,a][j+1];
12             c.predTr[x,Col_init[x,a][j+1]]=Col_init[x,a][j];
13          }
14    }
15 }
16
17 function firstInCollection(in c: Constraint; in x: Variable;
18   in a: Value):Index{
19     return c.FS[x,a];
20 }
21
22 function nextInCollection(in c:Constraint; in x: Variable;
23   in i: Index):Index{
24     return c.nextTr[x,i];
25 }
26
27 function isCollectionEmpty(in c: Constraint; in x: Variable;
28   in a: Value):Boolean{
29     return c.FS[x,a] == ⊤;
30 }
31
32 removeFromCollection(in c: Constraint; in x: Variable, in i: Index){
33    a = σ_{c,i}[x]
34    if(c.FS[x,a] == i){
35      c.FS[x,a] = c.nextTr[x,i];
36    } else if(i > c.FS[x,a]){
37      if (c.predTr[x,i]!=⊥)
38        c.nextTr[x,c.predTr[x,i],c] = c.nextTr[x,i,c];
39      if (c.nextTr[x,i]!=⊤)
40        c.predTr[x,c.nextTr[x,i],c] = c.predTr[x,i,c];
41    }
42 }
```

Algorithm 11: Specific Methods of AC5TCOpt-Tr

in backtrackable structure is $nextTr/predTr$. For a table constraint, along a branch in the search tree of length m, if k of its tuples are found Q-invalid, AC5TCOpt-Tr has $\Theta(r \cdot k)$ integers in its trailing record. Indeed, upon finding a Q-invalid tuple, for the r variables, either FS is modified, or the two pointers $nextTr$/$predTr$. This is much more than the previous algorithms. Along the same branch, they have $O(r \cdot k)$ integers to trail for FS and an additional k elements for AC5TC-Bool and at most m integer for AC5TC-Sparse for the Q-validity structures. AC5TC-Recomp only has to backtrack FS. As explained before, the worst case scenario leading to $O(r \cdot k)$ integers in the record for FS is not frequent.

5.2 AC5TCOpt-Sparse

The implementations of the specific methods of AC5TCOpt presented in the previous Section have one drawback: AC5TCOpt-Tr has to trail $\Theta(r \cdot t + r \cdot d)$ integers. Trailing such a large number of integers can be costly. By changing the data structures used to support the implementation of the specific AC5TCOpt methods, we can obtain a propagator trailing only $\Theta(r \cdot d)$ integers. This new implementation, called AC5TCOpt-Sparse, is the topic of this section.

The fundamental change to AC5TCOpt-Tr is to remove the order of the elements in the collections. Indeed, inside the $nextTr$ chains, the elements are ordered with respect to their order of appearance in the table. This is an unnecessary requirement. All that is needed is that those elements match the ones in the collections. We can therefore replace the $nextTr$ and $predTr$ structures with efficiently backtrackable sets. The structures we chose for representing those sets are an adapted version of Sparse Sets, introduced in Section 3.

Recall that the Sparse Set structure is using two different arrays for representing a set: Map and Dyn. In addition, Sparse Sets also maintain the size ($size$) of the set. Map contains, for each entry of the set, its position in the array Dyn. Dyn contains the elements of the set, before the index $size$ and the ones which have been removed, after $size$. There will be one such set per collection in Col, thus per literal. Each set will contain tuple indexes from Col. For a variable x and a tuple index i, $\sigma_{c,i}[x]$ is the only value for x. The index i can thus only be in one collection from Col, thus in one sparse set for x. This means that the Map array can be shared for all the literals of the same variable. Indeed, for a tuple i and a variable x, $Map[i]$ will represent the position of i in the Dyn array of the literal $(x, \sigma_{c,i}[x])$. The link between the arrays Map and Dyn can be formalized as:

$$\begin{aligned}&\forall x \in X, \forall i, j : 1 \leq i, j \leq t : \\ &\qquad Map[x][i] = j \Leftrightarrow Dyn[x, \sigma_{c,i}[x]][j] = i\end{aligned}$$

We will refer to this as the Dyn/Map invariant. Sharing the Map between all the values of a variable allows the size of the Map array to be $\Theta(t)$ (for each variable). After the creation of the structures, previously unseen elements are never added in Col. This allows the size of the Dyn array to be fixed to the initial number of support of its literal. Figure 5 shows an example of the structures for variable x. In this example, the arrays $Dyn[x, a]$ and $Dyn[x, b]$ can be used to traverse the collections. The array $Map[x]$ can be used, together with the table of the constraint, to locate an element in the different Dyn arrays in $O(1)$. Amongst those structures, only $size$ has to be trailed.

The implementation of the specific functions (Specification 2) of AC5TCOpt for AC5TCOpt-Sparse is given in Algorithm 12. Method `initSpecStruct` fills the Dyn, Map and $size$ structures. Method `firstInCollection` returns the first element in Dyn if the sparse set is not empty, $\top$ otherwise. As the elements are swapped in Dyn on removal, the first element is not always the one with the smallest index. Method `nextInCollection` returns the element following the current element in Dyn if it is at an index smaller than the $size$ of this set. It returns $\top$ otherwise. Method `removeFromCollection` swaps the element that is to be removed with the last one of the right Dyn array and updates the Map structure. This keeps the link between Map and Dyn consistent. This is the standard procedure for removing an element from a sparse set, adapted to the present case.

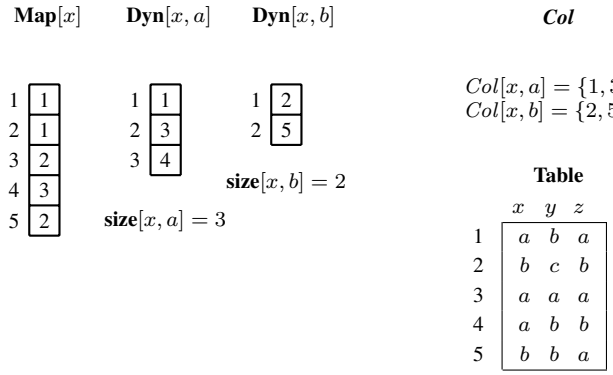


Fig. 5: Example of the Sparse Set structures used by AC5TCOpt-Sparse for variable x

Proposition 5 *The implementation of* `initSpecStruct`, `firstInCollection`, `nextInCollection`, `isCollectionEmpty` *and* `removeFromCollection` *for AC5TCOpt-Sparse is correct. The time complexity of* `initSpecStruct` *is* $O(r \cdot t + r \cdot d)$. *All the other methods are* $O(1)$. *The time complexity of AC5TCOpt-Sparse is the optimal* $O(r \cdot t + r \cdot d)$ *per table constraint along a branch in the search tree.*

Proof The correctness of the specific functions with respect to their individual specifications (Specification 2) is straightforward provided that Dyn and Map arrays are consistent. They are consistent if and only if the Dyn/Map invariant is respected. The invariant is trivially respected after the `initSpectStruct`. The only function that changes Dyn and Map after their creation is `removeFromCollection`. If the invariant is respected before `removeFromCollection`, it is respected after. The invariant is thus always respected. The functions `firstInCollection`, `nextInCollection` and the test `nextInCollection`$\neq \top$ thus form iterators over the collections in Col, the elements from $Col[x, a]$ being in $Dyn[x, a]$ between indexes 1 and $size[x, a]$. Recall that no remove operation is performed on a collection while iterating over it (hypothesis from Col). The $O(r \cdot t + r \cdot d)$ time complexity of `initSpecStruct` is granted by the $O(t)$ elements in the collections for all the values of a variable. ∎

Although the time complexity of AC5TCOpt-Sparse is the same as the one of AC5TCOpt-Tr (Section 5.1), the advantage of AC5TCOpt-Sparse is the number of integers to backtrack. AC5TCOpt-Sparse is backtracking only the $size$ array, which contains one integer per literal. It has thus only $\Theta(r \cdot d)$ integers to backtrack, compared to the $\Theta(r \cdot t + r \cdot d)$ integers that AC5TCOpt-Tr has to trail. More precisely, for a table constraint, on a branch in the search tree, if k tuples are found Q-invalid, AC5TCOpt-Sparse has $O(r \cdot k)$ integers on its trailing record. This worse case scenario corresponds to the situation where each tuple detected Q-invalid updates the collections of r different literals. Otherwise, only one integer is stored on the record when multiple updates of the same collection is performed at a node. For a table constraint, along a branch in the search tree detecting k tuples from the table to be Q-invalid, AC5TCOpt-Tr has a total number of integers in its trailing record of $\Theta(r \cdot k)$. AC5TCOpt-Sparse always has a number of integers in its trailing record that is less than or equal to the number in AC5TCOpt-Tr record, but most of the time, it has strictly fewer integers in its trail.

```
1  initSpecStruct(in c: Constraint, in Col_init: Matrix of Arrays){
2     forall(x in Vars(c), a in D(x)){
3        j=1;
4        forall(i in Col_init[x,a]){
5           c.Dyn[x,a][j] = i;
6           c.Map[x][i] = j;
7           j+=1;
8        }
9        c.size[x,a] = Col_init[x,a].length;
10    }
11 }
12
13 function firstInCollection(in c: Constraint; in x: Variable;
14   in a: Value){
15    if(c.size[x,a] > 0)
16       return c.Dyn[x,a][1];
17    else
18       return ⊤;
19 }
20
21 function nextInCollection(in c: Constraint; in x: Variable;
22   in i: Index){
23    if(c.Map[x][i] < c.size[x,a]){
24       return c.Dyn[x,σ_{c,i}[x]][c.Map[x][i]+1];
25    }else{
26       return ⊤;
27    }
28 }
29
30 function isCollectionEmpty(in c: Constraint; in x: Variable;
31   in a: Value){
32    return c.size[x,a] == 0;
33 }
34
35 removeFromCollection(in c: Constraint; in x: Variable, in i: Index){
36    a = σ_{c,i}[x];
37    c.Dyn[x,a][c.Map[x][i]] = c.Dyn[x,a][c.size[x,a]];
38    c.Dyn[x,a][c.size[x,a]] = i;
39    c.Map[x][c.Dyn[x,a][c.Map[x][i]]] = c.Map[x][i];
40    c.Map[x][i] = c.size - 1;
41    c.size-=1;
42 }
```

Algorithm 12: Specific Methods of AC5TCOpt-Sparse

The total space complexity of AC5TCOpt-Sparse is $\Theta(r \cdot t + r \cdot d)$. The size of Dyn is $\Theta(r \cdot t)$ since each tuple index is in exactly r Sparse Sets, the Map structure is $\Theta(r \cdot t)$ ($\Theta(t)$ for each variable) and the size structure is $\Theta(r \cdot d)$.

6 Experimental Results

All proposed algorithms have been implemented on top of Comet. For comparison, classical constraint-based algorithms have also been implemented on top of Comet. The AC3 and AC3rm algorithms [18], designed for binary constraints, have also been reimplemented. In

our implementation, these algorithms use a dichotomic search in the table to verify that a tuple is allowed by the constraint. The GAC3-Allowed algorithm has been chosen for the comparison because it is the standard GAC3 algorithm for non-binary table constraints [15]. The three existing state-of-the-art methods were also reimplemented: The MDD^c algorithm from [7], the STR2+ algorithm from [16] and STR3 from [19]. For MDD^c, the order of the variables can have a big impact on the performances of the algorithm. Indeed, the order strongly influences the size of the constructed MDD. Unfortunately, obtaining the perfect order is NP-Complete [7]. In the experiments, we used the variable order in the instances. For the STR2+ reimplementation, the array $lastSize$ is used. Some optimization can be obtained by reusing the structures constructed for a propagator between different constraints relying on the same table but with different scopes. This is the case, for instance, for the $next$ structure of our propagators or the MDD of MDD^c. This optimization has not been used in our test, for none of the propagators. All experiments were conducted on an Intel Xeon 2.53GHz using Comet 2.1.1. The algorithms are compared within a MAC search. The problems have been selected because they offer very different constraint arities. Some of them contain only binary tables while other contain up to arity 20 table constraints. This section thus presents results on the geometric problem, on Langford problem, on the Traveling Salesman Problem, on the RandRegular problem, on fully random instances, on Crosswords instances and on modified Renault instances. All the instances used are available on `http://becool.info.ucl.ac.be/resources`

For each instance set, the experimental results report the mean execution times in seconds (*totTime*), the mean "posting" times in seconds (*postTime*), the number of propagator calls (*nProp*), the percentage to the best with respect to execution time (*%best*), the mean of percentage to the best algorithm in terms of execution time (μ*%best*), the number of validity checks (*valChk*), Q-validity checks (*QvalChk*), and the number of pointers followed (*pFollow*). The difference between the *%best* and μ*%best* is the following: for *%best*, the execution times are averaged before computing the quantity. There is thus one best algorithm. For μ*%best*, the percentages are computed instance by instance and aggregated with a geometrical mean at the end. This measure takes into account that different instances may have different best algorithms. The $\mu\%best$ measure uses a geometrical mean as suggested in [9]. The last reported quantity, *pFollow*, has different meanings for different algorithms. For GAC3-Allowed, it corresponds to the number of times the tuples are accessed. For the AC5TC algorithms and AC5TCOpt-Tr, it is defined as the number of times the *next* or *nextTr* structures are used to traverse the table. For MDD^c, it corresponds to the number of edges followed in the MDD structure. Although referring to different quantities, *pFollow* is useful for comparing the behavior of the propagators as it reflects the usage of their specific structures. For each instance set, scatter plots of AC5TCOpt-Sparse versus STR2+ and STR3 are given. In those scatter plots, each point is an instance. The x axis of a point is the time taken by the algorithm on the bottom of the plot and the y axis, the time taken by the other algorithm. A point with coordinates (5,10) means that this instance has been solved in 5 seconds by the algorithm on the bottom and 10 by the other. The $x = y$ line is also displayed. The more points an algorithm has on its opponent side of the line $x = y$, the faster it is compared to the other. Those plots allow a detailed vision, instance by instance, of the performance of the algorithms. STR2+, STR3 and AC5TCOpt-Sparse have been chosen because AC5TCOpt-Sparse is the fastest of our algorithms and STR2+, STR3 are its best competitors. For the binary benchmarks, scatter plots of AC5TCOpt-Sparse versus AC3rm are also given.

The search strategy is given for each benchmark. We used the terminology defined in [1]. The *dom* variable heuristic chooses first the variable with the smallest domain. The *dom/deg* heuristic chooses first the variable with the smallest ratio domain size - degree of the variable (number of constraints in which it is involved). The *lexicographic* value ordering consists in trying first the smallest value with respect to the lexicographic ordering.

The Geometric Problem Instances of the geometric problem are random instances generated following a specific structure proposed by Rick Wallace [33]. Each variable is randomly placed in the unit square. A fixed distance (less than $\sqrt{2}$) is randomly chosen. For each pair of variables (x, y), if the distance between their associated points is less than or equal to this fixed distance, the arc (x, y) is added to the constraint graph. Constraint relations are then created as they are in fully random CSP instances [34]. The constraints of this problem are thus binary. We use the instance set from [14] which counts 100 instances. The search strategy uses the heuristic *dom/deg* with lexicographic value ordering. A timeout of 5 minutes has been used. The quantity *%solv* gives the percentage of solved instances.

propagator	totTime	postTime	nProp	%best	μ%best	%solv	valChk	QvalChk	pFollow
GAC3-Allowed	10.1	0.3	288 k	276	283	86	28 k	0	28 k
AC5TC-Bool	12.5	0.3	867 k	341	328	84	300	25 k	50 k
AC5TC-Sparse	10.8	0.2	867 k	295	271	86	300	25 k	50 k
AC5TC-Recomp	7.9	0.2	831 k	216	206	87	6 k	0	29 k
AC5TCOpt-Tr	9.6	0.8	867 k	263	412	87	300	0	13 k
AC5TCOpt-Sparse	6.5	0.4	867 k	178	236	87	300	0	0
MDD^c	14.7	1.6	288 k	401	694	86	0	0	65 k
STR2+	24.9	0.3	288 k	680	650	82	26 k	0	0
STR3	15.2	0.6	867 k	413	477	84	300	0	0
AC3	10.4	0.1	288 k	283	234	85	0	0	0
AC3rm	**3.7**	0.1	288 k	100	100	89	0	0	0

Table 1: Results of the propagators on the geom instances (times in seconds)

Table 1 presents the experimental results on geom instances. The quantities are computed on instances for which none of the techniques timeouts. All our propagators outperform the state-of-the-art STR2+, STR3 and MDD^c. AC5TCOpt-Tr, AC5Opt-Sparse and AC5TC-Recomp are also better than the classical AC3 and GAC3-Allowed propagators. AC3rm is clearly the winning strategy on those instances. It is also the best on each instance, as its $\mu\%best$ is 100. AC5TCOpt-Sparse is the fastest of our propagators on those instances. Its performances are competitive with AC3rm. It is however not the best of our propagators on each instance. The instances on which it is beaten are the smallest, where AC5TC-Recomp is the best of our propagators. AC5TCOpt-Sparse is significantly faster than AC5TCOpt-Tr, due to the cost that AC5TCOpt-Tr has to pay to trail its structures. Checking the validity (instead of the Q-validity) allows AC5TC-Recomp to follow less pointers than AC5TC-Bool and AC5TC-Sparse by performing longer jumps in the table. Moreover, as the tables are binary, the cost of validity check is low. AC5TCOpt-Tr follows far less pointers than AC5TC-Bool and AC5TC-Sparse because it does not follow pointers to a previously inspected tuple.

Scatter plots for the geom instance set are given in Figure 6. On those scatter plots, next to each technique name, are the number of instances that are solved by this algorithm that triggered a timeout with the other algorithm. For instance, AC5TCOpt-Sparse solved five instances that caused STR2+ to timeout. Those instances are not in the plot. As we can observe on those scatter plots, AC3rm is faster than AC5TCOpt-Sparse on all instances, except on the easiest ones. AC3rm solves 2 instances causing AC5TCOpt-Sparse to timeout. AC5TCOpt-Sparse solves more instances and is faster than STR2+ and STR3 on all the instances, except the easiest ones. We can also observe that STR3 is faster than STR2+. The time performances of those algorithms are proportional on this instance set.

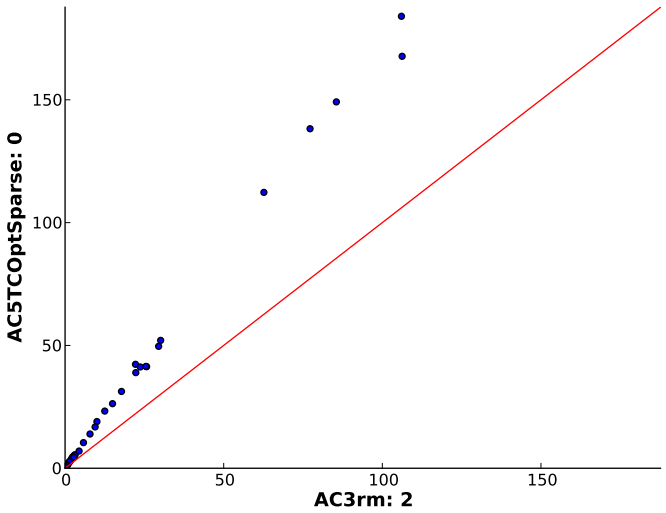
Langford Number Problem Langford number problem $L(k, n)$ amounts to arranging k sets of numbers 1 to n into a sequence of numbers, so that each occurrence of a number m is m numbers apart from its previous occurrence. This is problem 24 of CSPLIB [1]. Those problems can be modeled with binary (positive) table constraints only. The instances with table constraints can be found in [14]. The search strategy used is *dom/deg* with lexicographic value ordering. Problems where all the propagators take more than 5 minutes are removed from the sets. For $k = 2$, 12 instances are used: $n \in \{5..12, 15, 16, 19, 20\}$, for $k = 3$, 8 instances: $n \in \{3..10\}$ and for $k = 4$, 9 instances: $n \in \{3..11\}$. The results for k of 2, 3 and 4 can be found in Table 2.

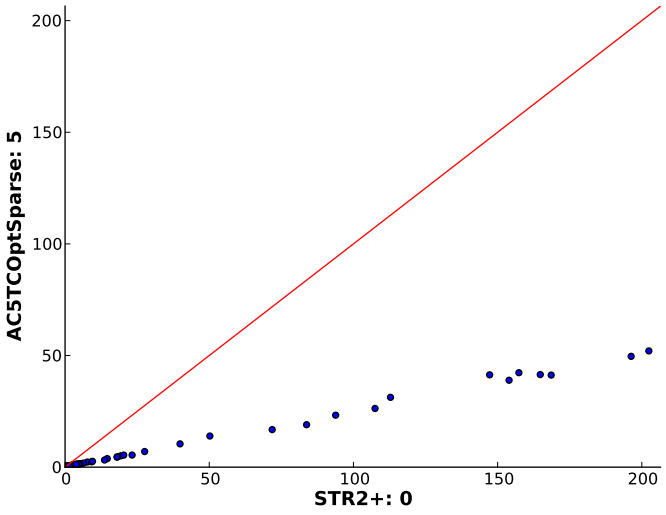
The winning strategy on those instances is clearly AC3rm. However, except for AC5TC-Bool on the $k = 4$ set of instances, all our propagators improve the state-of-the-art STR2+, STR3 and MDD^c. AC5TCOpt-Sparse, AC5TCOpt-Tr and AC5TC-Recomp are faster than AC3 on the $k = 2$ set. AC5TCOpt-Sparse and AC5TC-Recomp are faster than AC3 on the $k = 3$ set. The three fastest of our propagators on those instances are AC5TCOpt-Sparse, AC5TCOpt-Tr and AC5TC-Recomp. They are also better than the classical GAC3-Allowed. AC5TCOpt-Sparse is the fastest propagator on those three instance sets. Our optimal AC5TCOpt-Tr is faster than AC5TC-Recomp only for the $k = 2$ instance set. Observe that the number of followed pointers is globally higher for this instance set, due to inclusion of instances with larger n. The number of calls to the propagators during the search is also higher for the $k = 2$ set. This suggests that AC5TC-Tr requires harder instances (found in the $k = 2$ set) for amortizing the cost of its data structures. AC5TCOpt-Sparse does not have this problem thanks to its reduced need in backtrackable structures.

Scatter plots for the Langford problem are given for the $k = 4$ set in Figure 7. On those scatter plots are given, next to each algorithm, the number of instances that are solved by it and that caused the other algorithm to timeout. The scatter plots for the other values of k displayed the same patterns. The observations are similar than the ones made for the geom instance set: AC3rm seems linearly faster than AC5TCOpt-Sparse and AC5TCOpt-Sparse seems linearly faster than STR2+ and STR3. AC5TCOpt-Sparse and AC3rm are solving one instance more than STR2+ and STR3.

Traveling Salesman Problem We continue with results of the propagators on the Traveling Salesman Problem (TSP) constraint satisfaction instances. We used the set of instances *tsp-20* and *tsp-25* [14]. Those structured instances are composed of very different table constraints. Their arity varies between 2 and 3 and they may count up to 20 000 tuples but also as few as 20. The variables also have quite different domains: Some have small domains,

[1] www.csplib.org





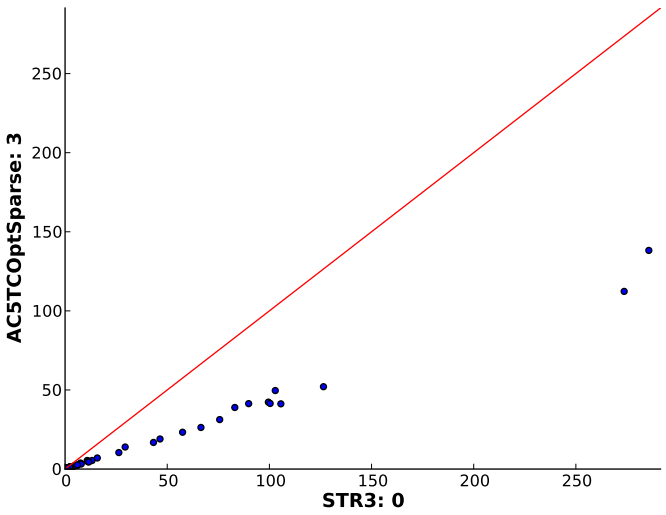
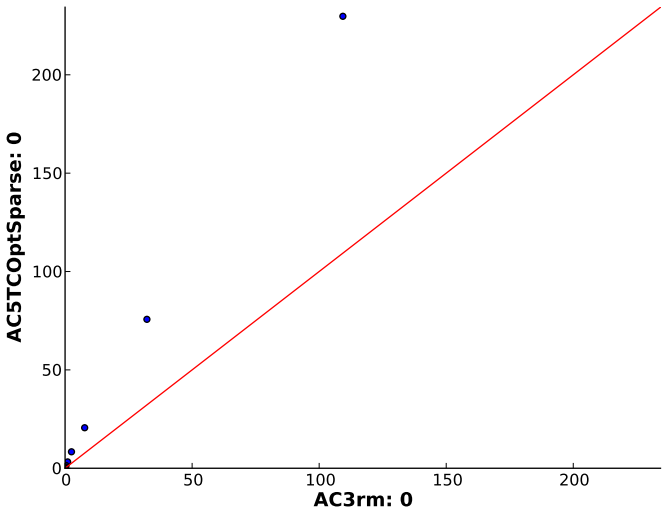
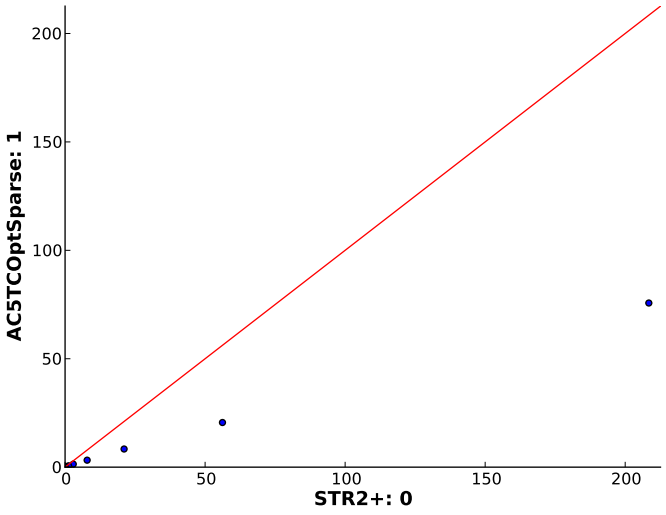


Fig. 6: Scatter plots of the Geom instance set





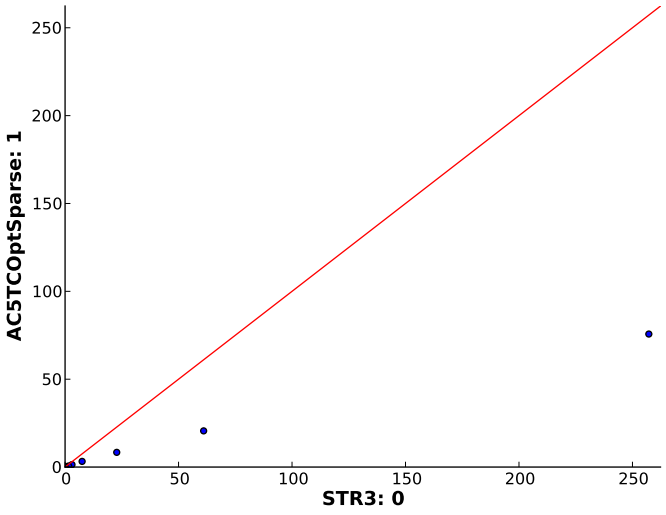


Fig. 7: Scatter plots of the Langford instance set for $k = 4$

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
				$k = 2$				
GAC3-Allowed	16.3	0.6	1 M	315	324	166 k	0	166 k
AC5TC-Bool	18.6	0.8	2 M	358	342	576	178 k	316 k
AC5TC-Sparse	16.8	0.5	2 M	323	276	576	178 k	316 k
AC5TC-Recomp	10.1	0.4	2 M	195	199	27 k	0	154 k
AC5TCOpt-Tr	9.4	2.5	2 M	182	488	576	0	42 k
AC5TCOpt-Sparse	6.6	0.8	2 M	126	264	576	0	0
MDD^c	26.6	3.7	1 M	512	970	0	0	307 k
STR2+	26.7	1.3	1 M	514	643	46 k	0	0
STR3	23.7	1.6	2 M	456	638	576	0	0
AC3	11.3	0.2	1 M	218	176	0	0	0
AC3rm	**5.2**	0.1	1 M	100	101	0	0	0
				$k = 3$				
GAC3-Allowed	2.5	0.3	75 k	395	310	12 k	0	12 k
AC5TC-Bool	3.5	0.3	242 k	553	395	380	10 k	21 k
AC5TC-Sparse	2.5	0.2	242 k	398	324	380	10 k	21 k
AC5TC-Recomp	1.5	0.2	239 k	244	213	2 k	0	12 k
AC5TCOpt-Tr	2.2	0.9	242 k	342	402	380	0	4 k
AC5TCOpt-Sparse	1.4	0.4	242 k	227	242	380	0	0
MDD^c	3.9	1.5	75 k	608	718	0	0	22 k
STR2+	3.7	0.6	75 k	585	546	5 k	0	0
STR3	4.0	0.7	242 k	639	627	380	0	0
AC3	1.6	0.1	85 k	223	183	0	0	0
AC3rm	**0.7**	0.1	85 k	100	100	0	0	0
				$k = 4$				
GAC3-Allowed	23.4	1.3	419 k	477	379	19 k	0	19 k
AC5TC-Bool	42.5	1.6	1.6 M	867	524	677	20 k	36 k
AC5TC-Sparse	29.8	1.0	1.6 M	608	384	677	20 k	36 k
AC5TC-Recomp	17.0	0.8	1.58 M	347	244	3 k	0	18 k
AC5TCOpt-Tr	21.8	5.0	1.6 M	445	621	677	0	5 k
AC5TCOpt-Sparse	12.3	1.7	2 M	250	315	677	0	0
MDD^c	31.2	7.3	419 k	637	957	0	0	35 k
STR2+	33.2	3.3	419 k	677	676	10 k	0	0
STR3	39.3	3.4	2 M	802	730	677	0	0
AC3	11.7	0.4	419 k	238	188	0	0	0
AC3rm	**4.9**	0.2	419 k	100	103	0	0	0

Table 2: Experimental Results on Langford instances (times in seconds)

while others feature domains containing up to 1000 values. There are 61 variables and 230 table constraints in *tsp-20* instances. The *tsp-25* instances count 76 variables and 350 constraints. The negative table constraints found in those instances have been transformed into positive ones. The search strategy used here is *dom/deg* with lexicographic value ordering. Both sets contain 15 instances. For the set *tsp-25*, instance *tsp-25-715* has been removed from the set, as it was unsolved after 3 hours.

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	797	1.7	6.7 M	1 073	795	11 M	0	11 M
AC5TC-Bool	186	0.8	21.2 M	251	254	2 k	1 M	2 M
AC5TC-Sparse	153	0.5	21.2 M	207	195	2 k	1 M	2 M
AC5TC-Recomp	109	0.3	20.9 M	146	140	391 k	0	1 M
AC5TCOpt-Tr	120	3.3	21.2 M	162	222	2 k	0	466 k
AC5TCOpt-Sparse	**74**	0.5	21 M	100	104	2 k	0	0
MDD^c	456	19.0	6.7 M	614	1041	0	0	7 M
STR2+	398	1.4	6.7 M	536	478	803 k	0	0
STR3	226	1.0	21 M	305	296	2k	0	0

Table 3: Results of the propagators for instance set TSP-20 (times in seconds)

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	6 607	2.4	73 M	931	764	23 M	0	23 M
AC5TC-Bool	2 625	1.3	198 M	370	350	2 k	11 M	19 M
AC5TC-Sparse	1 937	0.7	198 M	273	263	2 k	11 M	19 M
AC5TC-Recomp	1 315	0.5	196 M	185	180	3 M	0	10 M
AC5TCOpt-Tr	1 089	5.2	198 M	153	151	2 k	0	3 M
AC5TCOpt-Sparse	**710**	0.9	198 M	100	100	2 k	0	0
MDD^c	4 974	25.2	73 M	701	637	0	0	28 M
STR2+	3 740	2.9	73 M	527	500	5 M	0	0
STR3	2 308	1.9	198 M	325	305	2 k	0	0

Table 4: Results of the propagators for instance set TSP-25 (times in seconds)

Tables 3 and 4 present the results. We first observe that STR2+, STR3 and MDD^c perform worse than our propagators, except for the set TSP-25 where STR3 is faster than AC5TC-Bool. AC5TCOpt-Sparse is the winning strategy on both instance sets. It is the best for each instance of the TSP-20 set. Another observation is that AC5TC-Recomp is faster than our optimal AC5TCOpt-Tr on the TSP-20 set. On the contrary, AC5TCOpt-Tr is faster on the TSP-25 set. We can also see that checking the validity instead of the Q-validity allows AC5TC-Recomp to follow less pointers and perform fewer validity checks than the Q-validity checks of AC5TC-Bool and AC5TC-Sparse. Moreover, on these instances, the small arity makes the validity check ($O(r)$) cheap compared to Q-validity ($O(1)$). This explains the good performances of AC5TC-Recomp.

To test the effect of the arity on this instance set, we merged binary tables of the instances of the TSP-20 set into arity 4 tables. The merge is obtained by merging arity 2 constraints into arity 4 ones. The merged constraints do not share variables. The pruning in this benchmark is thus the same as in the original one. The results are summarized in Table 5. A timeout of 15 minutes has been set for those experiments. The data in the table concerns only instances for which none of the propagators timeouts. The percentage of the 15 instances solved by each propagator individually is also given. AC5TCOpt-Sparse is still the fastest propagator. Although, STR2+ and STR3 are faster than our other propagators on those instances. This seems to indicate that those existing state-of-the-art propagators are better for larger

arity constraints. The three propagators solving the largest number of instances are our two AC5TCOpt algorithms and STR2+.

Scatter plots for the TSP-20 and TSP-20 with quaternary tables are respectively given in Figure 8 and 9. The patterns on the set TSP-25 are similar to the ones in the scatter plots of the TSP-20 set. For the modified TSP-20 set, next to each algorithm, is the number of instances solved by this algorithm for which the other timeouts. As we can see on those scatter plots, AC5TCOpt-Sparse is linearly better than STR2+ and STR3 on both instance sets. STR2+ is a bit faster on the instance set with quaternary tables while STR3 is disadvantaged. Two instances with quaternary tables are solved by AC5TCOpt-Sparse while triggering a timeout for STR3.

propagator	totTime	postTime	nProp	%best	μ%best	%solved	valChk	QvalChk	pFollow
GAC3_Allowed	103	7.7	303 k	623	336	60	1 M	0	1 M
AC5TC-Bool	123	6.1	853 k	745	392	60	7 k	881 k	1 M
AC5TC-Sparse	99	4.0	853 k	600	303	67	7 k	881 k	1 M
AC5TC-Recomp	83	3.4	844 k	504	261	73	227 k	0	793 k
AC5TCOpt-Tr	60	44.1	853 k	362	429	93	7 k	0	36 k
AC5TCOpt-Sparse	**16.6**	7.3	852 k	100	102	93	7 k	0	0
MDD^c	130	104	303 k	782	934	87	0	0	456 k
STR2+	34.2	16.0	303 k	207	200	93	80 k	0	0
STR3	43.8	10.6	853 k	265	217	80	7 k	0	0

Table 5: Experimental Results on tsp-20 instances with arity 4 tables

RandRegular The Regular constraint [25] is a global constraint on a sequence of variable stating that the values taken by the variables have to form a word in a given regular language. The regular language is specified by a deterministic finite automaton. This constraint generalizes some other well known global constraints [25]. Examples of problems where those constraints are heavily used in CP are rostering problems. In rostering, the regular constraints are used to enforce the valid patterns of activities.

The regular constraints can be encoded efficiently with table constraints [2, 26]. Since the length of the sequence is fixed to the number of variables in the scope of the global constraint, additional variables can be introduced to represent the successive states visited in the automaton. For a regular constraint with scope $x_1 \dots x_r$, those state variables are $q_0 \dots q_r$. For all $0 \leq i < r$, a constraint links the variables in the scope and the additional variables: $q_{i+1} = \mathit{Trans}(q_i, x_{i+1})$, where Trans is the transition function of the automaton. Those constraints are posted using table constraints to encode the transition function. The tables are computed based on the transition function and the reachable states. Two additional constraints are posted: $q_0 = s$ and $q_n \in F$, where s is the starting state of the automaton and F is its set of final states.

For those experiments, we generated 100 instances with regular constraints. Those instances contain 20 regular constraints on 10 different variables. Each regular constraint has a scope of 5 variables, chosen randomly. The domains of the variable is of size 10. Each regular constraint is 20 states and has a randomly created transition table, hence the name of the

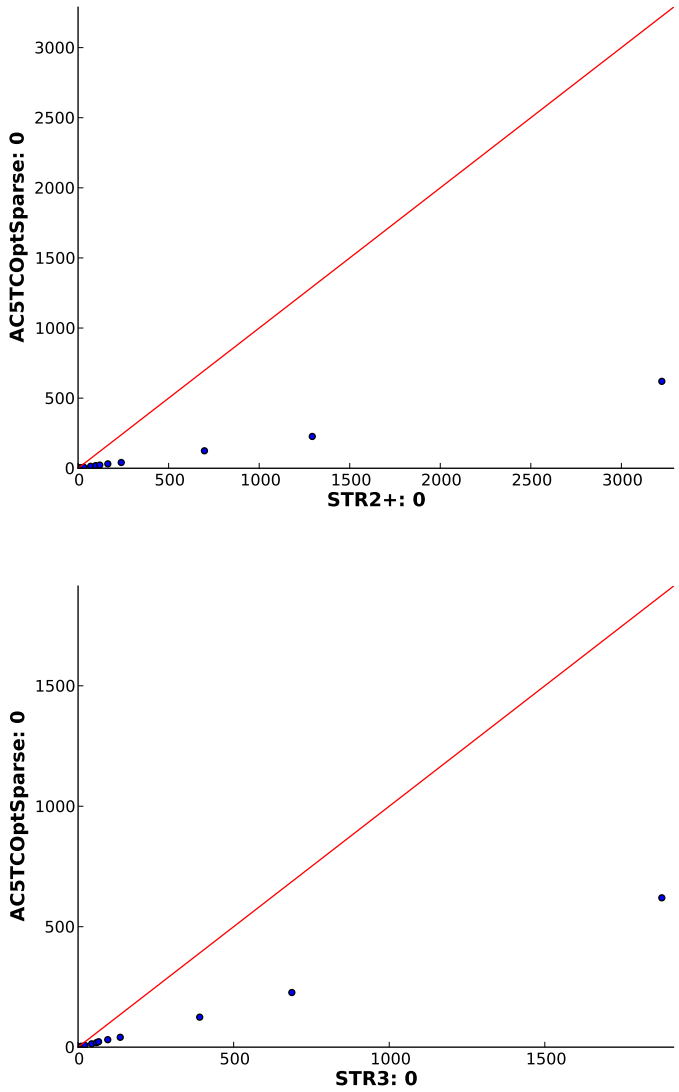


Fig. 8: Scatter plots of the TSP-20 instance set

benchmark: RandRegular. Amongst the states, 30 % of them are randomly chosen to be final. The parameters were chosen to produce instances with a significant number of fails and choice points. The regular constraints are transformed in ternary table constraints. The search strategy used during the resolution is *dom/deg* variable ordering with lexicographic variable ordering.

The results of the experiments on the RandRegular instances can be found in Table 6. On this instance set, all our propagators are faster than the existing state-of-the-art ones. The

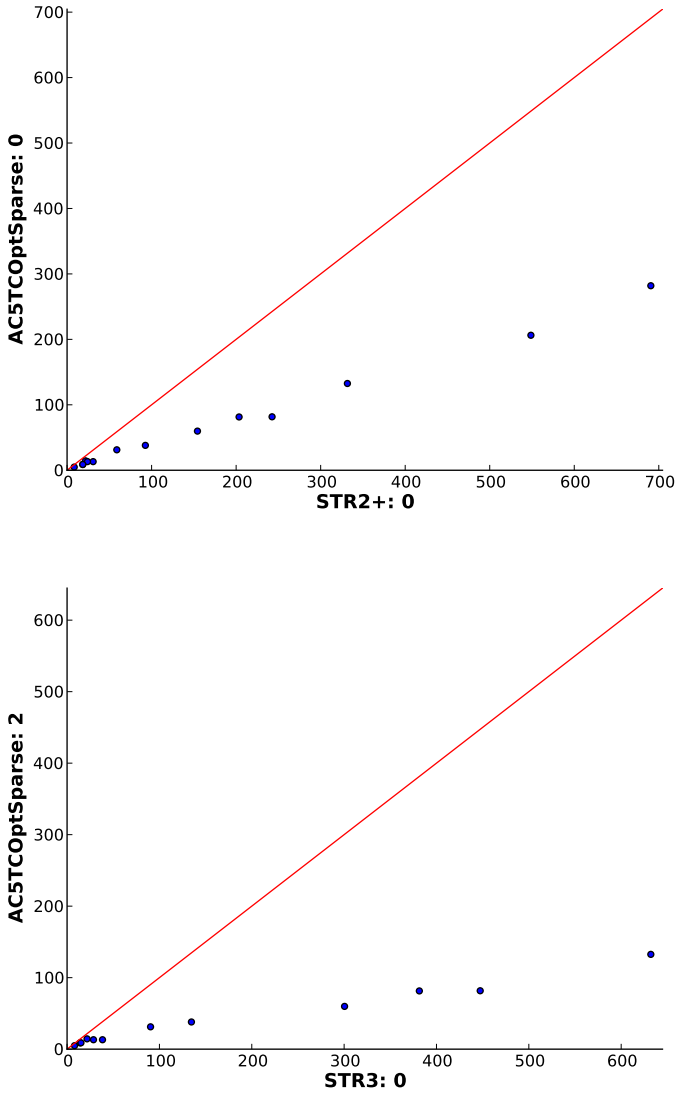


Fig. 9: Scatter plots of the modified TSP-20 instance set with quaternary tables

winning strategy is our optimal AC5TCOpt-Sparse. Despite the large variability of resolution times for a single technique between different instances, the small differences between the $\%best$ and $\mu\%best$ indicates that the performances of the propagators are proportional through the whole set. The arity of the tables (all tables have an arity of 3) as well as their medium size that is constant through the set could explain the good performances of our algorithms on this instance set.

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	68	0.1	2 M	205	200	1 M	0	1 M
AC5TC-Bool	40.5	0.0	8 M	122	121	137	462 k	955 k
AC5TC-Sparse	42.6	0.0	8 M	128	128	137	462 k	955 k
AC5TC-Recomp	36.4	0.0	8 M	110	109	288 k	0	766 k
AC5TCOpt-Tr	42.5	0.1	8 M	128	129	137	0	367 k
AC5TCOpt-Sparse	**33.2**	0.1	8 M	100	100	137	0	0
MDD^c	105	0.6	2 M	316	313	0	0	1 M
STR2+	96	0.0	2 M	288	283	371 k	0	0
STR3	69	0.1	8 M	208	207	137	0	0

Table 6: Experimental Results on RandRegular Instances

Scatter plots for the RandReg benchmark are given in Figure 10. We can observe that the solving times are well spread for those instances. We can also see that AC5TCOpt-Sparse is linearly faster than both STR2+ and STR3, STR3 being faster than STR2+.

Random Instances These instances contain random table constraints of random scope generated by the RD-model [34]. Parameters are chosen to generate instances close to the phase transition, using Theorems 1 and 2 from [34]. The instances have 10 variables, a uniform domain size of 10, and 15 table constraints of arity 5. The expected number of tuples in each table is thus 20000. 10 instances were generated with those settings. The search strategy is the *dom* heuristic with lexicographic value ordering.

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	3 000	1.5	614 k	2 725	2 660	523 M	0	523 M
AC5TC-Bool	4 636	1.0	2.8 M	4 211	4 070	19 k	257 M	481 M
AC5TC-Sparse	3 991	0.8	2.8 M	3 626	3 538	19 k	257 M	481 M
AC5TC-Recomp	3 874	0.8	2.4 M	3 519	3 357	98 M	0	305 M
AC5TCOpt-Tr	994	5.2	2.8 M	903	930	19 k	0	16 M
AC5TCOpt-Sparse	469	1.7	2.8 M	426	440	19 k	0	0
MDD^c	**110**	12.4	614 k	100	100	0	0	12 M
STR2+	483	0.7	614 k	439	455	22 M	0	0
STR3	913	2.1	2.8 M	829	839	19 k	0	0

Table 7: Results of the propagators on fully random instance set (times in seconds)

Table 7 summarizes the results. Results are similar for other parameter settings which also generate instances close to the phase transition. The standard MDD^c algorithm outperforms our value-based propagators on all instances, as it has a $\mu\%best$ of 100. The performance of our optimal AC5TCOpt-Tr is comparable to the performance of the optimal STR3 but AC5TCOpt-Sparse is significantly faster than both. AC5TCOpt-Sparse is the most efficient value based propagator. Observe the large number of validity checks of AC5TC-Recomp and Q-validity checks of AC5TC-Bool and AC5TC-Sparse, as well as the number of times they follow a pointer. The two AC5TCOpt implementations are performing the same number of validity checks at post time as the two AC5TC ones but they do not require any

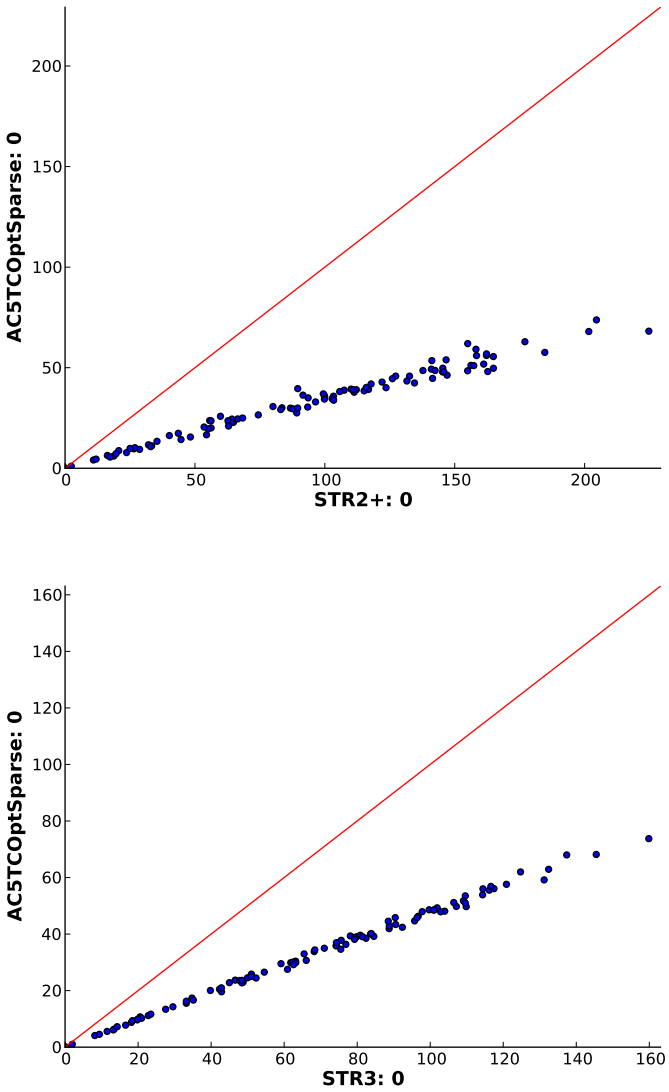


Fig. 10: Scatter plots of the RandRegular instance set

Q-validity checks afterwards. The difference in performance between AC5TCOpt-Tr and AC5TCOpt-Sparse is reflecting the additional cost AC5TCOpt-Tr has to pay for trailing its larger data structures. The overall performances of our propagators seems to be impacted by the augmentation of the arity of the tables as well as their size. Indeed, AC5TC-Recomp, AC5TC-Sparse and AC5TC-Bool can revisit each tuple r time in the worst case and the trailable structures of AC5TCOpt-Tr are proportional to $r \cdot t$.

Figure 11 shows the scatter plots of AC5TCOpt-Sparse versus STR2+ and AC5TCOpt-Sparse versus STR3. On those graphs, we can see that the performance of STR2+ and AC5TCOpt-Sparse are similar. On the other side, AC5TCOpt-Sparse is faster than STR3.

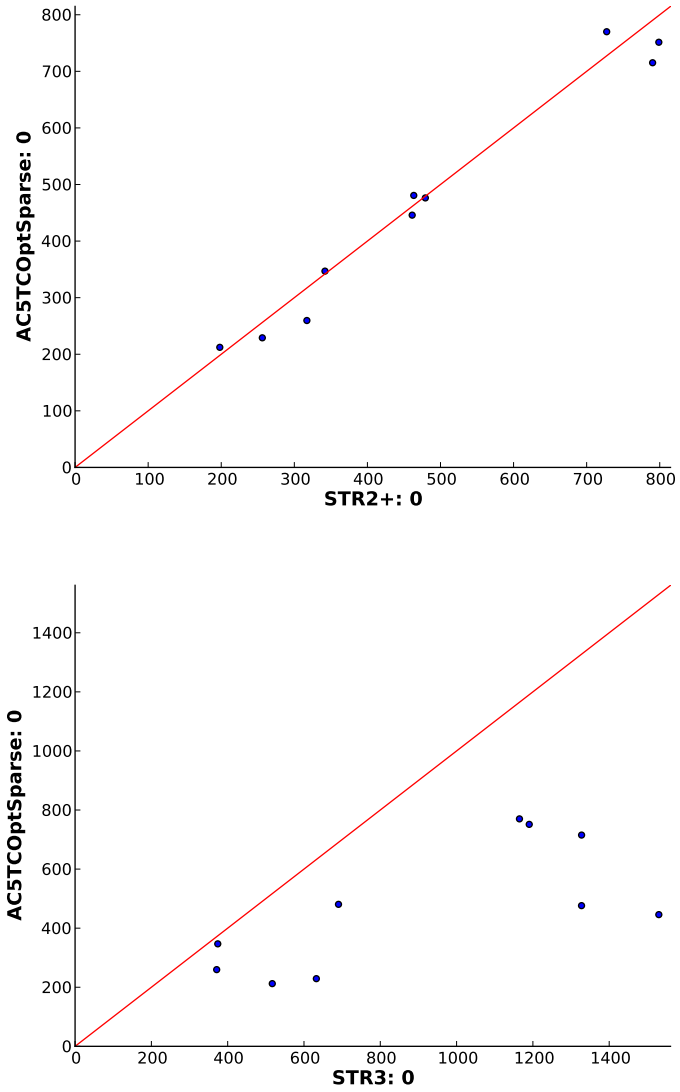


Fig. 11: Scatter plots of the Random Benchmark

Crosswords Problem The Crosswords problem is the problem of filling a predefined grid with words from a dictionary. We used four instance sets from [14]. Those instances are also

used to test table constraint propagators in [16] and [19]. The instances in those sets differ by the dictionary they use to get the words from. The grids are all the same and empty. Inside a set, different grid sizes are used, varying the arity of the table constraints. The instance set *lexVg* is using the dictionary defined in [29]. Instances in *ogdVg* use a french dictionary. The set *ukVg* is using the UK cryptic solvers dictionary. The last instance set, *wordsVg* uses the dictionary in /usr/dict/words under Linux. The dictionaries of *lexVg* and *wordsVg* instances sets have a rather small dictionary, leading to small tables. On the contrary, *ogdVg* and *ukVg* have large dictionaries. As, for those problems, the same word can be used multiple times, only table constraints are used to encode the problem.

The search heuristic used for this problem is *dom/deg* variable ordering with lexicographic value ordering. A timeout of 20 minutes has been set on the resolution of the instances. The results concerns only the instances for which none of the propagators timeouts. The grid sizes used in the experiments are:

- for *lexVg*, 42 instances: 4x{4..8}, 5x5, 6x6, 7x{10..11}, 8x{9..12}, 9x{9..13}, 10x{10..14}, 11x{11..15}, 12x{12..16}, 13x{13..17}, 14x{14..18}, 15x{15..18}, 16x{16..20}
- for *ogdVg*, 27 instances: 4x{4..8}, 5x{5..9}, 6x6, 7x7, 13x{16..17}, 14x{16..18}, 15x{15..19}, 16x{16..20}
- for *ukVg*, 23 instances: 4x{4..8}, 5x{5..7}, 6x{6}, 13x17, 14x{16..18}, 15x{15..19}, 16x{16..20}
- for *wordsVg*, 47 instances: 4x{4..8}, 5x{5..7}, 6x6, 7x11, 8x{11..12}, 9x{10..13}, 10x{10..14}, 11x{11..15}, 12x{12..16}, 13x{13..17}, 14x{14..18}, 15x{15..17}, 16x{16..18}

The four sets contain a total of 139 instances. The arities of the tables is determined by the grid size. An instance with grid size $x \times y$ has table constraints of arity x and y.

The results of the propagators on the Crosswords instances can be found in Table 8. Except for AC5TCOpt-Sparse on the *lexVg* set, both STR3 and STR2+ are faster than all other propagators. Although, the $\mu\%best$ quantity indicates that neither of them is the best on each instance. When compared to AC5TCOpt-Sparse, they are faster on the easiest and the hardest instances but AC5TCOpt-Sparse is faster on the medium difficulty ones. On the sets *lexVg* and *ukVg*, the two AC5TC propagators have smaller $\mu\%best$ than STR2+ and STR3, meaning that they are generally closer to the best instance by instance. Another observation is that, although AC5TCOpt-Tr is optimal, the non optimal AC5TC-Bool and AC5TC-Sparse are faster on the four sets of crossword instances. On the *lexVg* and the *wordsVg* sets, AC5TCOpt-Sparse is the best of our propagators. Surprisingly, on the *ogdVg* and *ukVg*, the best of our propagator is AC5TC-Sparse, followed by AC5TC-Bool. Those two instance sets are the ones where the dictionaries used are the bigger, meaning tables with more tuples. AC5TCOpt-Tr seems particularly disadvantaged by its large backtrackable structures on this problem. The characteristic of this benchmark, with large arity tables, seems to disadvantage our propagators.

Scatter Plots for *ogdVg* and *wordsVg* are given respectively in Figures 12 and 13. The integer next to each algorithm is the number of instances solved by it triggering a timeout for the other algorithm. Those two instance sets have been chosen because *ogdVg* instances have a large dictionary and *wordsVg* instances have a small one. For *ogdVg*, STR2+ is faster than AC5TCOpt-Sparse except on some instances where their performance is comparable. STR3 is clearly faster than AC5TCOpt-Sparse except on three non-easy instances where AC5TCOpt-Sparse is significantly faster than STR3. AC5TCOpt-Sparse is able to solve one

propagator	totTime	postTime	nProp	%best	μ%best	valChk	QvalChk	pFollow
				lexVg				
GAC3-Allowed	61	0.2	35 k	234	283	9 M	0	9 M
AC5TC-Bool	44.9	0.1	611 k	171	150	2 k	5 M	7 M
AC5TC-Sparse	41.2	0.1	611 k	157	139	2 k	5 M	7 M
AC5TC-Recomp	50	0.1	543 k	192	170	4 M	0	6 M
AC5TCOpt-Tr	66	0.6	611 k	252	378	2 k	0	847 k
AC5TCOpt-Sparse	30.3	0.2	611 k	116	191	2 k	0	0
MDD^c	77	7.8	35 k	293	1321	0	0	3 M
STR2+	31.8	0.2	35 k	121	160	957 k	0	0
STR3	**26.2**	0.2	611 k	100	167	2 k	0	0
				ogdVg				
GAC3-Allowed	55	2.9	4 k	264	263	6 M	0	6 M
AC5TC-Bool	41.5	1.2	61 k	200	168	12 k	3 M	4 M
AC5TC-Sparse	37.7	1.0	61 k	182	151	12 k	3 M	4 M
AC5TCRecomp	53	0.9	51 k	254	182	2 M	0	4 M
AC5TCOpt-Tr	122	14.7	61 k	589	563	12 k	0	348 k
AC5TCOpt-Sparse	52	2.8	61 k	249	212	12 k	0	0
MDD^c	146	82	4 k	704	1703	0	0	1 M
STR2+	33.6	1.6	4 k	162	180	548 k	0	0
STR3	**20.7**	3.1	61 k	100	151	12 k	0	0
				ukVg				
GAC3-Allowed	76	1.3	14 k	244	290	6 M	0	6 M
AC5TC-Bool	67	0.6	373 k	216	243	7 k	4 M	6 M
AC5TC-Sparse	58	0.5	373 k	186	212	7 k	4 M	6 M
AC5TC-Recomp	83	0.4	292 k	267	275	3 M	0	5 M
AC5TCOpt-Tr	175	4.3	373 k	565	482	7 k	0	506 k
AC5TCOpt-Sparse	77	1.2	372 k	247	203	7 k	0	0
MDD^c	222	56	14 k	713	1110	0	0	3 M
STR2+	41.9	0.6	14 k	135	128	583 k	0	0
STR3	**31.1**	1.3	373 k	100	150	7 k	0	0
				wordsVg				
GAC3-Allowed	63	0.3	20 k	246	296	8 M	0	8 M
AC5TC-Bool	49.5	0.2	362 k	193	161	2 k	5 M	6 M
AC5TC-Sparse	43.9	0.1	362 k	171	148	2 k	5 M	6 M
AC5TC-Recomp	56	0.1	317 k	219	173	3 M	0	5 M
AC5TCOpt-Tr	88	1.1	362 k	345	466	2 k	0	734 k
AC5TCOpt-Sparse	39.6	0.3	362 k	155	207	2 k	0	0
MDD^c	84	11.8	20 k	328	1840	0	0	2 M
STR2+	35.4	0.3	20 k	138	180	921 k	0	0
STR3	**25.6**	0.4	362 k	100	169	2 k	0	0

Table 8: Experimental Results on Crosswords instances

instance STR3 can't within the time limit. On *wordsVg*, the performances of STR2+ and AC5TCOpt-Sparse are comparable, STR2+ being faster in average. STR3 is also faster than AC5TCOpt-Sparse on this instance set (except for two non-easy instances) but the relative difference is smaller here than in *ogdVg*. AC5TCOpt-Sparse is able to solve one *wordsVg* instance causing both STR2+ and STR3 to timeout.

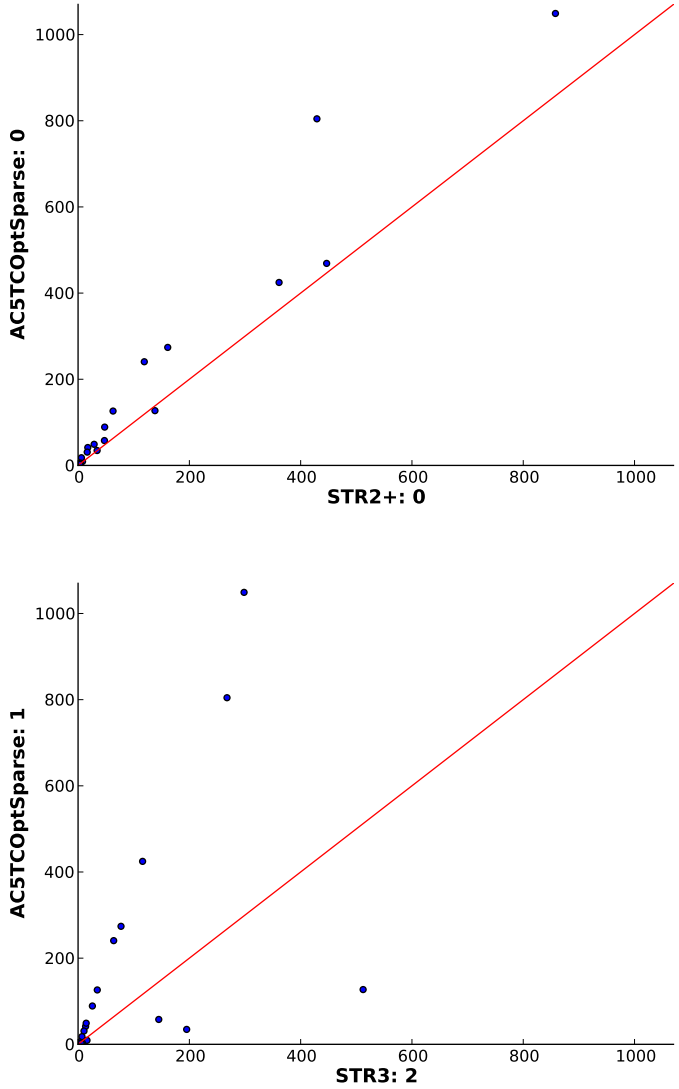


Fig. 12: Scatter plots of the crossword instance set *ogdVg*

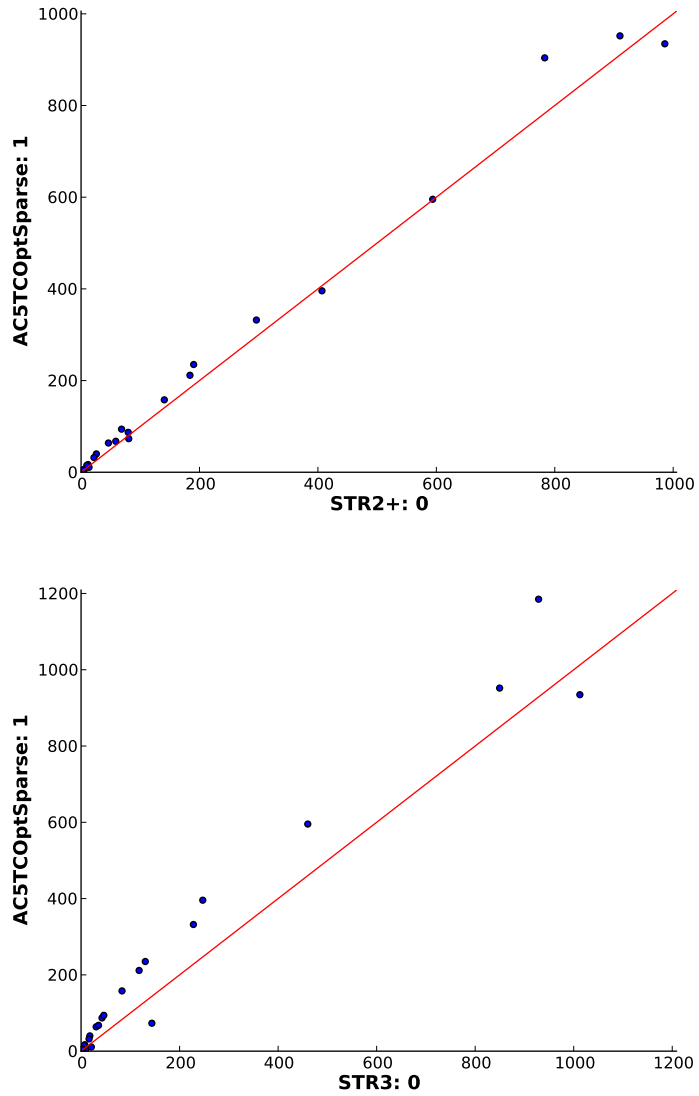


Fig. 13: Scatter plots of the crossword instance set *wordsVg*

Modified Renault Problem The modified renault problem instances originate from a Renault Megane configuration problem. This problem has been modified in order to generate a series of instances. Those instances have the particularity to present large tables (up to 50 k tuples) of large arities (up to arity 10). Those instances can be found in [14]. The search strategy used is *dom/deg* variable ordering with lexicographic value ordering. A timeout of 20 minutes has been set on the resolution time. The results concern only the instances for which none of the propagators timed out. The set of instances found on [14] counts 50 instances.

Amongst them, 16 are solved by all propagators within the time limit. The percentage of those 50 instances solved by each individual propagator is also given in the result table.

The experimental results on the Modified Renault Problem are given in Table 9. STR2+ is the winning strategy on this instance set and it is the fastest on each instance. Our optimal AC5TCOpt-Sparse is faster than STR3 and it is solving one instance more. However, STR3 is faster than our other propagators. Observe the difference in the number of calls to the propagator between the value based and constraint based propagators, giving advantage to the constraint based approaches. The difference in the number of calls between AC5TCOpt-Sparse and the group AC5TCOpt-Tr, AC5TC-Sparse and AC5TC-Bool comes from the order in which the tuples are visited during propagation. Indeed, all our propagators visit the tuples in the order of the table, except AC5TCOpt-Sparse. This results in a difference in the order of the values in the propagation queue and hence, a difference in the number of calls to the propagators. AC5TC-Recomp has less calls due to its use of the validity, allowing it to compute a larger $\triangle$. Although the large arity of the tables seems to slow our propagators, AC5TCOpt-Sparse is impacted the less.

propagator	totTime	postTime	nProp	%best	μ%best	%solved	valChk	QvalChk	pFollow
GAC3-Allowed	16.5	1.4	39 k	307	215	34	286 k	0	286 k
AC5TC-Bool	14.4	0.8	268 k	268	202	32	1 k	128 k	221 k
AC5TC-Sparse	13.2	0.6	268 k	244	178	32	1 k	128 k	221 k
AC5TC-Recomp	12.1	0.5	227 k	224	174	32	67 k	0	155 k
AC5TCOpt-Tr	14.5	4.5	268 k	269	389	36	1 k	0	33 k
AC5TCOpt-Sparse	7.6	1.6	275 k	141	176	36	1 k	0	0
MDD^c	17.9	15.4	39 k	332	642	36	0	0	21 k
STR2+	**5.4**	0.6	39 k	100	100	36	25 k	0	0
STR3	10.4	1.7	268 k	193	198	34	1 k	0	0

Table 9: Experimental Results on the modified Renault problem

Scatter plots for those instances can be found in Figure 14. Next to each algorithm is the number of instances solved by it that the other algorithm was unable to solve within the time limit. Although the spreading of the solving time is not good, those plots confirm the tendency exhibited by the average solving time in the table: STR2+ faster than AC5TCOpt-Sparse and AC5TCOpt-Sparse faster than STR3. We can also observe that AC5TCOpt-Sparse solves one instance that STR3 is unable to solve within the time limit.

Summary Table 10 gives a summary of the per benchmark percentage to the best mean time. The names of our propagators have been shortened by removing the prefix 'AC5TC'. This table shows the effect of the arity of the table constraints on the propagators. The benchmarks are separated into three categories depending on the arity of the table constraints: the binary benchmarks, the small arity benchmarks (arity 3 and 4) and the large arity benchmarks. For the benchmarks containing only binary table constraints, AC3rm is clearly the fastest propagator. However, on those benchmarks, our propagators are globally faster than the existing state-of-the-art MDD^c, STR2+ and STR3. AC3rm has been designed for binary constraints. For the benchmarks where the tables have arity up to 4, our propagators are globally the best propagators. However, when the arity of the tables in the benchmarks

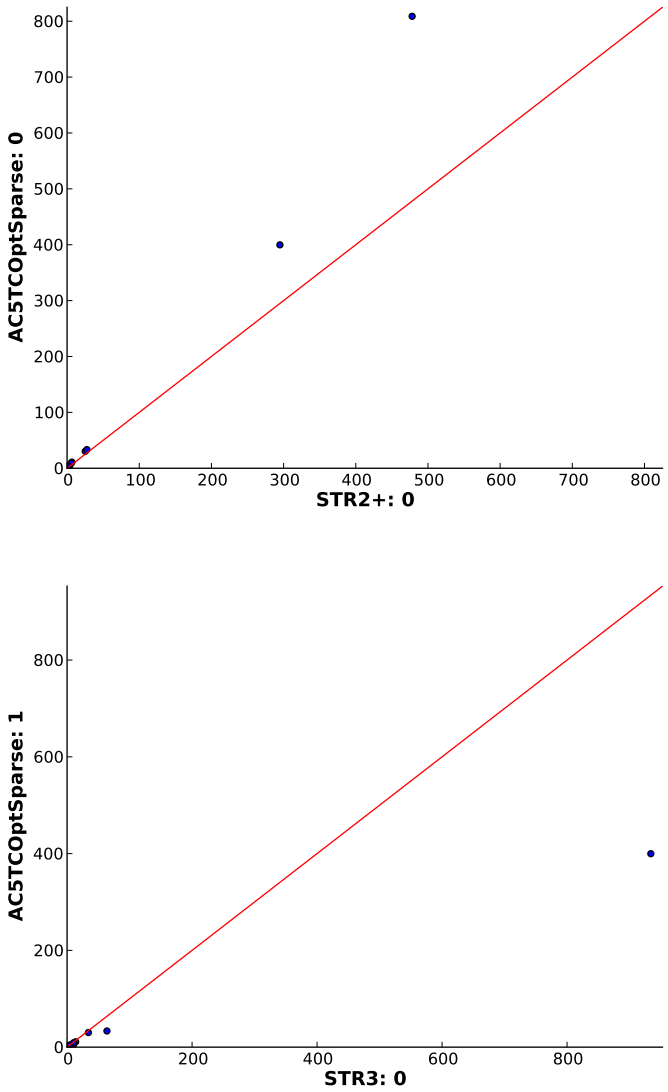


Fig. 14: Scatter plots of the modified Renault instance set

increases, our propagators become slower than the state of the art. The existing state-of-the-art propagators considered in this paper are well suited for problems where the arity is large. The MDD^c propagator is the fastest on the random instances. On this set, despite the random tables, its compressed table is small, allowing it to outperform the other propagators. The optimal STR3 propagator is the fastest on the crosswords instances. STR2+ is the fastest on the modified Renault benchmark. Although it would be statistically meaningless to average the $\%best$ in Table 10, it is however clear that, on these benchmarks,

our optimal AC5TCOpt-Sparse is our best propagator. It stays competitive with AC3rm on half of the binary benchmarks, it is globally the best on the small arity benchmarks and on some of the large arity benchmarks, its performances are competitive with the state of the art. The non optimal AC5TC-Recomp and the optimal propagator AC5TCOpt-Tr are the next fastest ones of our propagators. AC5TC-Tr outperforms AC5TC-Recomp on difficult instances. However, on easier instances, the cost of its trailable *nextTr* data-structure makes it slower than AC5TC-Recomp. AC5TC-Bool and AC5TC-Sparse are generally slower than AC5TC-Recomp since they are testing Q-validity, not validity, and hence perform smaller jumps in the table. However, on the crossword instances, the two AC5TC algorithms are faster than AC5TC-Recomp on all instance sets and even than AC5TCOpt-Sparse on two of the four sets. Also, AC5TCOpt-Sparse is always faster than AC5TCOpt-Tr, confirming the additional cost AC5TCOpt-Tr has to pay for its backtrackable structures. AC5TC-Sparse is also always faster than AC5TC-Bool, for the same reasons.

Benchmark	GAC3_Allowed	Bool	Sparse	Recomp	OptTr	OptSparse	MDD^c	STR2+	STR3	AC3	AC3rm
geom	276	341	295	216	263	178	401	680	413	283	**100**
Langford(2)	315	358	323	195	182	126	512	514	456	218	**100**
Langford(3)	395	553	398	244	342	227	608	585	639	223	**100**
Langford(4)	477	867	608	347	445	250	637	677	802	238	**100**
TSP-20	1 073	251	207	146	162	**100**	614	536	305	-	-
TSP-25	931	370	273	185	153	**100**	701	527	325	-	-
TSP-Quat-20	623	745	600	504	362	**100**	782	207	265	-	-
RandRegular	205	122	128	110	128	**100**	316	288	208	-	-
Random	2 725	4 211	3 626	3 519	903	426	**100**	439	829	-	-
CW-LexVg	234	171	157	192	252	116	293	121	**100**	-	-
CW-ogdVg	264	200	182	254	589	249	704	162	**100**	-	-
CW-ukVg	244	216	186	267	565	247	713	135	**100**	-	-
CW-wordsVg	246	193	171	219	345	155	328	138	**100**	-	-
modified Renault	307	268	244	224	269	141	332	**100**	193	-	-

Table 10: Summary of the experimental results: %best

7 Conclusion

This paper proposed five different value-based, domain-consistency algorithms for table constraints, all using the AC5 generic framework. The new propagators record, for every value of the variables, the index of its first current support in the table. They also use, for each variable of a tuple, the index of the next tuple sharing the same value for this variable. They differ in their use of information on the validity of the tuples as well as the order of the tuples in the formed next chains. AC5TCOpt-Sparse is globally the best of our value-based algorithms. As AC5TCOpt-Tr, AC5TCOpt-Sparse embeds the Q-validity information into the indexing structure, avoiding unnecessary visits of invalid tuples and leading to an optimal algorithm with a time complexity of $O(r \cdot t + r \cdot d)$ per table constraint. However, AC5TCOpt-Sparse relaxes the requirement that the tuples in the structure are ordered as they are in the table. Doing that allows AC5TCOpt-Sparse to have far less backtrackable structures. While not changing its theoretical complexity, this relaxation allows AC5TCOpt-Sparse to be more

efficient in practice. Our other algorithms have a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint. Experimental results show that, on instances containing only binary table constraints, our algorithms are outperformed by AC3rm. However, they are faster than STR2+, STR3 and MDD^c. When the arity of the tables in the instances is up to 4, our propagators are the fastest ones. The speedups that our propagators provide on those benchmarks are up to 5.36 over STR2+, up to 7.82 over MDD^c and 3.25 over STR3. However, when the arity of the table increases, the conclusion changes. The existing state-of-the-art propagators STR2+, STR3 and MDD^c are faster than our algorithms on one different benchmark each. As future work, it would be interesting to extend AC5TC to handle negative tables through its disallowed tuples and to integrate the compressed representation of tuples introduced in [27].

Acknowledgments The authors want to thank the anonymous reviewers for their constructive and helpful comments. Thanks to Christophe Lecoutre for a discussion on the complexity of GAC3-allowed and GAC2001-allowed. The first author is supported as a Research Assistant by the Belgian FNRS (National Fund for Scientific Research). This research is also partially supported by the Interuniversity Attraction Poles Program (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS. This work was conducted in part at NICTA and is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. van Beek P (2006) Backtracking search algorithm. In: [28]
2. Beldiceanu N, Carlsson M, Debruyne R, Petit T (2005) Reformulation of global constraints based on constraints checkers. Constraints 10(4):339–362
3. Bessière C (2006) Constraint propagation. In: [28]
4. Bessière C, Régin JC (1997) Arc consistency for general constraint networks: Preliminary results. In: IJCAI (1), pp 398–404
5. Briggs P, Torczon L (1993) An efficient representation for sparse sets. ACM Letters on Programming Languages and Systems (LOPLAS) 2(1-4):59–69
6. Carlsson M (2006) Filtering for the case constraint, talk given at the advanced school on global constraints
7. Cheng K, Yap R (2010) An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints 15:265–304
8. Deville Y, Van Hentenryck P (2010) Domain consistency with forbidden values. In: Proceedings of CP 2010, Springer, pp 191–205
9. Fleming PJ, Wallace JJ (1986) How not to lie with statistics: the correct way to summarize benchmark results. Commun ACM 29(3):218–221
10. Gange G, Stuckey PJ, Szymanek R (2011) Mdd propagators with explanation. Constraints 16(4):407–429

11. Gent IP, Jefferson C, Miguel I (2006) Watched literals for constraint propagation in Minion. In: Proceedings of CP 2006, Springer-Verlag, pp 182–197

12. Gent IP, Jefferson C, Miguel I, Nightingale P (2007) Data structures for generalised arc consistency for extensional constraints. In: Proceedings of the AAAI 07, AAAI Press, pp 191–197

13. Katsirelos G, Walsh T (2007) A compression algorithm for large arity extensional constraints. In: Proceedings of CP 2007, Springer-Verlag, pp 379–393

14. Lecoutre C Instances of the Constraint Solver Competition. http://www.cril.fr/∼lecoutre/

15. Lecoutre C (2009) Constraint Networks: Techniques and Algorithms. ISTE/Wiley

16. Lecoutre C (2011) Str2: optimized simple tabular reduction for table constraints. Constraints 16:341–371

17. Lecoutre C, Szymanek R (2006) Generalized arc consistency for positive table constraints. In: Proceedings of CP 2006, pp 284–298

18. Lecoutre C, Hemery F, et al (2007) A study of residual supports in arc consistency. In: Proceedings of IJCAI 2007, vol 7, pp 125–130

19. Lecoutre C, Likitvivatanavong C, Yap R (2012) A path-optimal gac algorithm for table constraints. In: Proceedings of ECAI 2012, pp 510–515

20. Lhomme O (2004) Arc-consistency filtering algorithms for logical combinations of constraints. In: Proceedings of CPAIOR 2004, pp 209–224

21. Lhomme O, Régin JC (2005) A fast arc consistency algorithm for n-ary constraints. In: Proceedings of the Nationnal Conference on Artificial Intelligence, AAAI Press, pp 405–410

22. Mairy J.B., Van Hentenryck P., Deville Y. (2012) An Optimal Filtering Algorithm for Table Constraints. In: Proceedings of CP 2012, vol 3709, Springer Berlin, pp 496–511.

23. Mohr R, Masini G (1988) Good old discrete relaxation. In: Proceedings of ECAI 1988, pp 651–656

24. Perron L, Furnon V or-tools, http:// code.google.com/p/or-tools

25. Pesant G (2004) A regular language membership constraint for finite sequences of variables. In: Proceedings of CP 2004, Springer, pp 482–495

26. Quimper CG, Walsh T (2006) Global grammar constraints. In: Proceedings of CP 2006, Springer, pp 751–755

27. Régin JC (2011) Improving the expressiveness of table constraints. In: In proceedings of ModRef 2011 Workshop held with CP 2011

28. Rossi F, Beek Pv, Walsh T (eds) (2006) Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA

29. Samaras N, Stergiou K (2005) Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. Journal of Artificial Intelligence Research 24(1):641–684

30. Ullmann JR (2007) Partition search for non-binary constraint satisfaction. Information Sciences 177(18):3639–3678

31. Van Hentenryck P, Ramachandran V (1995) Backtracking without Trailing in CLP($\Re_{lin}$). ACM Transactions on Programming Languages and Systems 17(4):635–671

32. Van Hentenryck P, Deville Y, Teng CM (1992) A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57(2-3):291–321

33. Wallace R (2005) Factor analytic studies of csp heuristics. In: Proceedings of CP 2005, vol 3709, Springer Berlin / Heidelberg, pp 712–726

34. Xu K, Boussemart F, Hemery F, Lecoutre C (2007) Random constraint satisfaction: Easy generation of hard (satisfiable) instances. Artificial Intelligence 171(8-9):514–534