An Optimal Filtering Algorithm for Table Constraints

Jean-Baptiste Mairy[1], Pascal Van Hentenryck[2], and Yves Deville[1]

[1] ICTEAM, Université catholique de Louvain, Belgium
{Jean-Baptiste.Mairy, Yves.Deville}@uclouvain.be
[2] Optimization Research Group, NICTA, University of Melbourne, Australia
pvh@nicta.com.au

Abstract. Filtering algorithms for table constraints are constraint-based, which means that the propagation queue only contains information on the constraints that must be reconsidered. This paper proposes four efficient value-based algorithms for table constraints, meaning that the propagation queue also contains information on the removed values. One of these algorithms (AC5TC-Tr) is proved to have an optimal time complexity of $O(r.t + r.d)$ per table constraint. Experimental results show that, on structured instances, all our algorithms are two or three times faster than the state of the art STR2+ and MDD^c algorithms.

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). For table constraints, which have been the focus of much research in recent years, all existing algorithms (except in [14]) are constraint-based. This paper proposes four 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. Three of the proposed algorithms have a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint and one of them (AC5TC-Tr) has 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 One of the proposed algorithms, AC5TC-Recomp, is the (unpublished) propagator of the Comet system.

Experimental results show that, on structured instances, our algorithms improve upon the state-of-the-art STR2+ [11] and MDD^c [3]: The speedup is between 1.95 and 3.66 over STR2+ and between 1.83 and 4.57 over MDD^c. Our (theoretically) optimal algorithm is not always the fastest in practice. Interestingly, on purely random tables, our algorithms do not compete with STR2+ and MDD^c. Since most real problems are structured, 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 presents our optimal propagator, while Section 5 proposes an efficient variant of our first algorithms. 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 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 $D(Y)$. A support in a constraint c for a variable value pair (x, a) is a tuple $\mathbf{v} \in D(Vars(c))$ such that $c(\mathbf{v})$ and $\mathbf{v}[x] = a$. The following (Inconsistent and Consistent) sets are useful for specifying domain consistency and propagation methods. Let c be a constraint of a CSP $(X, D(X), C)$ with $y \in Vars(c)$, and $B(X)$ be some domain.

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

A constraint c in a CSP $(X, D(X), C)$ is *domain-consistent* iff $Inc(c) = \emptyset$. 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 $(x_1, \ldots, x_r) \in T$. The size t of a table constraint c is its number of tuples, which is denoted by $c.length$. 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 value $\top$ (resp. bottom value $\bot$) greater (resp. smaller) than any other value. 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 implies that, 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. As usual for such algorithms, if a domain-consistency algorithm has a time complexity of $O(f)$, then the time complexity of the aggregate executions of this algorithm along any path in the search tree is also $O(f)$.

Related Work. A lot of research effort has been spent on table constraints. The existing propagators can be categorized in 3 classes: index-based, compression-based, and based on a dynamic table. 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) [10,1,12,6]. 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 is $O(r^3 \cdot d \cdot t + r \cdot d^2)$ per table constraint. GAC2001_allowed has a time complexity of $O(r^3 \cdot d \cdot t + r^2 \cdot t)$ per table constraint. Indexing can also be used in value-based propagators. In [14], 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 [13]. Another index type, proposed in [7], 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 [7]. Another example of compression-based techniques [3,2] 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. Compression and faster traversal can also be achieved by using compressed tuples, which represent a set of tuples [8,16]. Propagators based on dynamic tables maintain the table by suppressing invalid tuples from it. The STR algorithm [18] and its refined version, STR2 [11], are constraint-based and scan only the previously valid tuples to extract the valid values. The time complexity of STR2 is $O(r^2 \cdot d^2 + r^2 \cdot d \cdot t)$ per table constraint. The *or-tool* propagator [15] also maintains a dynamic table. It uses a bitset on the tuples of the table to maintain their validity. One bitset per variable value (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.

The AC5 Algorithm. AC5 [19,4] is a generic value-based domain-consistency algorithm. In a value-based approach, information on the removed values is also stored in the queue for the propagation. Specification 1 describes the main methods of AC5 which 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 $D(x)$. When a value is removed from a domain, the method `enqueue` puts the necessary information on the queue. In the postcondition, Q_o represents the value of Q at call time. The method `post`(c,$\triangle$) is called once when posting the constraint. It computes the inconsistent values of the constraint c and initializes specific data structures required for the propagation of the constraint. 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 table constraints, a tuple σ is ***Q-valid*** if all its values belong to $D(X, Q, c)$. 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; inout Q: Queue)
// Pre: $x \in X$, $a \notin D(x)$
// Post: Q = $Q_0 \cup \{(c, x, a) | c \in C, x \in Vars(c)\}$
post(in c: Constraint;out △: Set of Values)
// Pre: $c \in C$
// Post: △ = $Inc(c)$ + initialization of specific data structures
valRemove(in c: Constraint; in y: Variable; in b: Value;
          out △: Set of Values)
// Pre: $c \in C$, $b \notin D(y, Q, c)$
// Post: $\triangle_1 \subseteq \triangle \subseteq \triangle_2$ with $\triangle_1 = Inc(c, D(X, Q, c)) \cap Cons(c, y, b)$
//                          and $\triangle_2 = Inc(c)$
```

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

3 Efficient Value-Based Algorithms for Table Constraints

Our value-based approaches 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). 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. The $next$ data structure is semantically equivalent to the index of [12]. More formally, for a given table constraint c, *FS* and *next* satisfy the following invariant (called FS-invariant) before dequeuing an element from Q.

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

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 1 and 2. They use the `seekNextSupportTC` method (Algorithm 3) which searches the next Q-valid tuple. Abstract method `isQValidTC(c,i)` tests whether $\sigma_{c,i}$ is Q-valid (i.e., $\sigma_{c,i} \in D(X, Q, c)$) 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 $FS[y, b]$. When one of these 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. If such a support does not exist, then (x, a) belongs to the set $\triangle_1$. Method `valRemoveTC` thus computes the set $\triangle_1$ and maintains the FS-invariant.

Not considering `initSpecStructTC`, method `postTC` has a time complexity of $O(rt + rd)$. After the `postTC` method, the domain size of x is $O(t)$. We now establish the complexity of all executions of `valRemoveTC` for a given table constraint,

```
postTC(in $c$: Constraint;out $\triangle$: Set of Values){
// Pre: $c \in C$, $c$ is a table constraint
// Post: $\triangle$ = $Inc(c)$ + initialization of the next, FS and specific data structures
    $\triangle$ = $\emptyset$;
    initSpecStructTC($c$);
    forall($x$ in $Vars(c)$, $a$ in $D(x)$) $c$.FS[$x$,$a$]=$\top$;
    forall($x$ in $Vars(c)$, $i$ in 1..$c.length$)  $c$.next[$x$,$i$] = $\top$;
    forall($i$ in $c.length$..1)
        if ($\sigma_{c,i}$ in $D(Vars(c))$){
            forall(x in $Vars(c)$){
              $c$.next[$x$,$i$] = FS[$x$,$\sigma_{c,i}[x]$];
              $c$.FS[$x$,$\sigma_{c,i}[x]$] = $i$;
            }
        }
        else setQInvalidTC($c$,$i$);
    forall($x$ in $Vars(c)$,$a$ in $D(x)$)
        if($c$.FS[$x$,$a$]==$\top$) $\triangle$ += (x,a);
}
```
Algorithm 1: Method `postTC` for Table Constraints

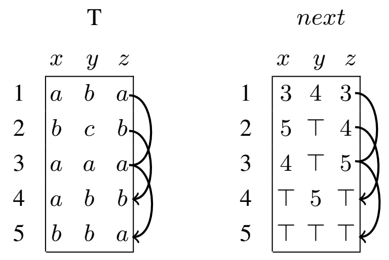
assuming this table constraint is one of the constraints of the CSP on which domain consistency is achieved. Consider first all executions of `valRemoveTC` without line 13. For a given variable y, these executions follow the different $next$ chains of the variable y. The chains for all values of y have a total number of t elements. The complexity of lines 9–16 (without line 13) is $O(r)$. Since the table has r variables, the complexity of all `valRemoveTC` executions during the fixed point (without line 13) is thus $O(r^2 \cdot t)$, assuming a $O(1)$ complexity of `setQInvalidTC`. Consider now all executions of line 13 in `valRemoveTC` for a variable x. Since line 13 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 13 in `valRemoveTC` thus take time $O(V \cdot r \cdot t)$. The time complexity of all executions of `valRemoveTC` is then $O(r^2 \cdot t + V \cdot r \cdot t)$. Even with a $O(1)$ the time complexity of `isQValidTC`, the algorithm is not optimal but it turns out to be more efficient than state-of-the-art algorithms on different classes of problems. 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.*

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 4. It uses a data structure *isQValid*$[i]$ to record the Q-validity of the element $\sigma_{c,i}$. It satisfies invariant

```
valRemoveTC(in c: Constraint;in $y$: Variable; in $b$: Value;
            out $\triangle$: Set of Values) {
// Pre: $c \in C$, $c$ is a table constraint and $b \notin D(y,Q,c)$
// Post: $\triangle_1 \subseteq \triangle \subseteq \triangle_2$ with $\triangle_1 = Inc(c, D(X,Q,c)) \cap Cons(c,y,b)$
//                       and $\triangle_2 = Inc(c,x)$
    $\triangle$ = $\emptyset$;
    $i$ = $c$.FS[$y,b$];
    while($i$!=$\top$){
        setQInvalidTC($c,i$);
        forall($x$ in $Vars(c)$: $x$!=$y$){
            $a$ = $\sigma_{c,i}[x]$;
            if ($c$.FS[$x,a$]==$i$){
                $c$.FS[$x,a$] = seekNextSupportTC($c,x,i$);
                if($c$.FS[$x,a$]==$\top$ && $a$ in $D(x)$)   $\triangle$ += $(x,a)$;
            }
        }
        $i$ = $c$.next[$y,i$];
    }
}
```

Algorithm 2: Method `valRemoveTC` for Table Constraints.



	T		
	x	y	z
1	a	b	a
2	b	c	b
3	a	a	a
4	a	b	b
5	b	b	a

	$next$		
	x	y	z
1	3	4	3
2	5	$\top$	4
3	4	$\top$	5
4	$\top$	5	$\top$
5	$\top$	$\top$	$\top$

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

isQValid$[i] \Leftrightarrow \sigma_{c,i} \in D(X,Q,c)$ before dequeuing an element from Q ($1 \leq i \leq c.length$). The data structure must be trailed as it depends on the domains. The methods for Q-validity are given in Algorithm 4. 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 and STR2 [18,11]. The implementation simply keeps invalid elements at the end of the table, with a single variable $size$ representing the boundary between valid (before position $size$) and invalid elements (after position $size$). When an element becomes invalid, it is swapped with the element at position $size$ and $size$ is decremented by one. The $size$ variable must be trailed but the table does not need to: The valid elements are automatically restored, albeit at a different position in the table. This is sometimes called semantic backtracking [20]. Instead of swapping tuples, our implementation uses two arrays

```
function seekNextSupportTC(in c: Constraint; in x: Variable;
                           in i: Index)  : Index {
// Pre: $c \in C$, c is a table constraint, $x \in Vars(c)$, $1 \leq i \leq c.length$
// Post: return the first index greater than i which is Q−valid
    i = c.next[x,i];
    while(i!=⊤){
        if(isQValidTC(c,i))  return i;
        i = c.next[x,i];
    }
    return ⊤;
}
```

Algorithm 3: Function `seekNextSupportTC` for Table Constraints.

```
initSpecStructTC-Bool(in c: Constraint) {
    forall(i in 1..c.length)  c.isQValid[i] = true;
}
function isQValidTC-Bool(in c: Constraint;in i: Index) {
// Pre: $c \in C$, c is a table constraint and $1 \leq i \leq c.length$
// Post: returns $\sigma_{c,i} \in D(X, Q, c)$
    return c.isQValid[i];
}
setQInvalidTC-Bool(in c: Constraint;in i: Index) {
// Pre: $c \in C$, c is a table constraint and $1 \leq i \leq c.length$
    c.isQValid[i] = false;
}
```

Algorithm 4: Implementation of the specific methods of AC5TC-Bool

Map and *Dyn* that give the virtual position of the tuples and the positions of the virtual tuples in the table. These arrays do not have to be trailed. For a given table constraint c, the data structures satisfy the following invariants before dequeuing an element from Q ($1 \leq i \leq c.length$): $Map[i] \leq size \Leftrightarrow \sigma_i \in D(X, Q, c) \ \wedge \ Dyn[Map[i]] = i$

The implementations are given in Algorithm 5 and the algorithm is called AC5TC-CutOff. The time complexity of `isQValidTC-CutOff` and `setQInvalidTC-CutOff` is $O(1)$ and `initSpecStructTC-CutOff` takes time $O(t)$. The time complexity of AC5TC-CutOff is thus $O(r^2 \cdot t + r \cdot d)$.

4 An Optimal Algorithm

In method `valRemoveTC`, executions of the `seekNextSupportTC` (line 12 of Algorithm 2) take $O(r \cdot t)$ assuming `isQValidTC` takes constant time. However, the method revisits Q-invalid tuples because the *next* data structure is static. To remedy this situation, the idea is to make the $next$ data structure dynamic and to always ensure that the element following a Q-valid element in a $next$ chain is also Q-valid. This avoids unnecessary Q-validity checks and can be easily implemented using a doubly-linked list.

```
initSpecStructTC-CutOff(in c: Constraint){
   forall(i in 1..c.length){
      c.Map[i] = i;
      c.Dyn[i] = i;
   }
   c.size = c.length;
}
function isQValidTC-CutOff(in c: Constraint; in i: Index;out b: Bool){
// Pre: $c \in C$, c is a table constraint and $1 \leq i \leq c.length$
// Post: return ($\sigma_{c,i} \in D(X, Q, c)$)
    return (c.Map[i] <= c.size);
}
setQInvalidTC-CutOff(in c: Constraint; in i: Index){
// Pre: $c \in C$, c is a table constraint and $1 \leq i \leq c.length$
   c.Dyn[c.Map[i]] = c.Dyn[c.size];
   c.Dyn[c.size[c]] = i;
   c.Map[c.Dyn[c.Map[i]]] = c.Map[i];
   c.Map[i] = c.size;
   c.size--;
}
```
Algorithm 5: Implementation of the specific methods of AC5TC-CutOff

More formally, for a given table constraint c, the data structure satisfies the following invariant before dequeuing an element from Q

$$\begin{aligned}
&\forall x \in \mathit{Vars}(c)\ \forall 1 \leq i \leq c.length :\\
&\qquad \mathit{nextTr}[x,i] = Min\{j | i < j \wedge \sigma_{c,j}[x] = \sigma_{c,i}[x]\\
&\qquad\qquad \wedge (\sigma_{c,i} \in D(X,Q,c) \Rightarrow \sigma_{c,j} \in D(X,Q,c))\}\\
&\qquad \mathit{predTr}[x, \mathit{nextTr}[x,i]] = i
\end{aligned}$$

The *nextTr* and *predTr* data structures should be trailed as they depend on the current domains. The algorithm also uses the FS data structure with its original invariant. No other data structures are necessary.

Methods `postTC-Tr` and `valRemoveTC-Tr` are given in Algorithms 6 and 7. Method `postTC-Tr` now initializes *predTr* as well. Method `valRemoveTC-Tr`(c, y, b) does not need to search for a support as the next element in the *nextTr* chain is necessarily Q-valid. However, if the first support for (x, a) is before *FS*$[y, b]$ (it cannot be after because of the FS invariant), *nextTr* and *predTr* must be updated to ensure that the new invalid tuples are no longer in the $next$ chains. It will thus never be visited twice. Method `valRemoveTC-Tr` computes the set $\triangle_1$ and maintains the invariants on FS and on $nextTr$ / $predTr$.

Method `postTC-Tr` has a time complexity of $O(r \cdot t + r \cdot d)$. We establish the complexity of all executions of `valRemoveTC-Tr` for a given table constraint during the fixed point algorithm, assuming the presence of other constraints on which domain consistency is also enforced. We first show that all executions of lines 7 and 21 lead to

```
postTC-Tr(in c: Constraint; out △: Set of Values){
// Pre: $c \in C$, c is a table constraint
// Post: △ = Inc(c) + initialization of the next, pred and FS data structures
    △ = ∅;
    forall(x in Vars(c), a in D(x))  c.FS[x,a]=⊤;
    forall(x in Vars(c), i in 1..c.length)
        c.nextTr[x,i] = ⊤; c.predTr[x,i] = ⊥;
    forall(i in c.length..1: $\sigma_{c,i}$ in D(Vars(c)))
        forall(x in Vars(c)){
            c.nextTr[x,i] = c.FS[x,$\sigma_{c,i}$[x]];
            if (c.FS[x,$\sigma_{c,i}$[x]]!=⊤) c.predTr[x,FS[x,$\sigma_{c,i}$[x]] = i;
            c.FS[x,$\sigma_{c,i}$[x]] = i;
        }
    forall(x in Vars(c),a in D(x))
        if(c.FS[x,a]==⊤) △ += (x,a);
}
```

Algorithm 6: An optimal `postTC-Tr` method for Table Constraints

different values of i in $\{1, \ldots, t\}$ (except when $i == \top$). By the FS invariant, we never have $i == \top$ at line 7. For a given value $i \neq \top$, by the FS invariant, $FS[x, a] == i$ or $FS[x, a] < i$ holds at line 11. If $FS[x, a] == i$, $FS[x, a]$ is incremented and the tuple i will never be reconsidered by removing a from $D(x)$. If $FS[x, a] < i$, the tuple i is removed from the $nextTr$ chain and will never be reconsidered. This holds for all variables $x \neq y$. Hence the tuple i will never be reconsidered in future executions of `valRemoveTC-Tr`. Hence, lines 9-20 are executed $O(t)$ times. Since the complexity of lines 9-20 is $O(r)$, the aggregate complexity of all executions of `valRemoveTC-Tr` is $O(r \cdot t)$. The AC5 algorithm with the `postTC-Tr` and `valRemoveTC-Tr` implementation for table constraint is called AC5TC-Tr (AC5 for Table Constraints with Trailing).

Proposition 2. *AC5TC-Tr is correct and has an optimal time complexity of $O(r \cdot t + r \cdot d)$ per table constraint.*

5 A Variation Based on Recomputation

We now propose a variation of the AC5TC algorithm, called AC5TC-Recomp, that does not require any data structure to maintain the Q-validity of tuples. AC5TC-Recomp is the (unpublished) table constraint algorithm of the Comet system. It replaces the Q-validity test by a function `isValidTC` that tests the validity of the tuples. The straightforward implementation of the `isValid` function is given in Algorithm 8. Method `initSpecStructTC` and `setQInvalidTC` are just empty. Since AC5TC-Recomp tests validity instead of Q-validity, method `valRemoveTC` must be slightly modified; the test $a \in D(x)$ should be moved from line 14 to line 10 which becomes[3]
`forall(`x `in` $Vars(c)$`:` x`!=`y `&&` $\sigma_{c,i}[x]$ `in` $D(x)$`){`

[3] This modification also maintains the correctness of our generic AC5TC algorithm but requires a more sophisticated FS-invariant. With this change, our earlier algorithms would have the same theoretical complexity but are less efficient in practice.

```
valRemoveTC-Tr(in c: Constraint; in $y$: Variable; in $b$: Value;
               out $\triangle$: Set of Values) {
// Pre: $c \in C$, $c$ is a table constraint and $b \notin D(y, Q, c)$
// Post: $\triangle_1 \subseteq \triangle \subseteq \triangle_2$ with $\triangle_1 = Inc(c, D(X, Q, c)) \cap Cons(c, y, b)$
//                         and $\triangle_2 = Inc(c)$
  $\triangle$ = $\emptyset$;
  $i$ = $c$.FS[$y$,$b$];
  while($i$!=$\top$){
    forall($x$ in $Vars(c)$: $x$!=$y$){
      $a$ = $\sigma_{c,i}[x]$;
      if($c$.FS[$x$,$a$] == $i$){
        $c$.FS[$x$,$a$] = $c$.nextTr[$x$,$i$];
        if($c$.FS[$x$,$a$]==$\top$ && $a$ in $D(x)$)   $\triangle$ += ($x$,$a$);
      } else { //$c$.FS[$x$, $a$] < $i$
        if ($c$.predTr[$x$,$i$]!=$\perp$)
          $c$.nextTr[$x$,$c$.predTr[$x$,$i$],$c$] = $c$.nextTr[$x$,$i$,$c$];
        if ($c$.nextTr[$x$,$i$]!=$\top$)
          $c$.predTr[$x$,$c$.nextTr[$x$,$i$],$c$] = $c$.predTr[$x$,$i$,$c$];
      }
    }
    $i$ = $c$.next[$y$,$i$];
  }
}
```
Algorithm 7: An Optimal `valRemoveTC-Tr` method for Table Constraints

This modified version of `valRemoveTC` exploits the flexibility of its specification by computing a set $\triangle$ between $\triangle_1$ and $\triangle_2$. AC5TC-Recomp has a runtime complexity of is $O(r^2 \cdot t + r \cdot d)$ and per table constraint and improves state-of-the-art algorithms on some classes of problems.

6 Experimental Results

All proposed algorithms have been implemented on top of Comet, including AC5TC-Recomp. For comparison, classical constraint-based algorithms have also been implemented on top of Comet. The GAC3-Allowed algorithm has been chosen because it is the standard GAC3 algorithm for the table constraints [10]. The two state-of-the-art methods were also reimplemented: The STR2+ algorithm from [11] and the MDD^c algorithm from [3]. They are respectively called STR and MDD in the experimental results. All experiments were conducted on an Intel Xeon 2.53GHz using Comet 2.1.1. The algorithms are compared within a MAC search. This section presents results on fully random instances, on the geometric problem, on Langford problem, and on the Traveling Salesman Problem.

For each instance set, the experimental results report the mean execution times in seconds (*totTime*), the mean "posting" times in seconds (*postT*), 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 num-

```
function isValid(in c: Constraints;in i: Index) : Bool {
// Pre: $c \in C$, $c$ is a table constraint and $1 \leq i \leq c.length$
// Post: return ($\sigma_{c,i} \in D(X)$)
   forall($x$ in $Vars(c)$)
      if(!$\sigma_{c,i}[x]$ in $D(x)$) return false;
   return true;
}
```
Algorithm 8: The `isValid` Function of AC5TC-Recomp.

ber 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 [5]. 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, it is defined as the number of times the *next* or *nextTr* structures are used to traverse the table. For MDD, 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.

Random Instances These instances contain random table constraints of random scope generated by the RD-model [22]. Parameters are chosen to generate instances close to the phase transition, using Theorems 1 and 2 from [22]. 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	postT	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-CutOff	3 991	0.8	2.8 M	3 626	3 538	19 k	257 M	481 M
AC5TC-Tr	994	5.2	2.8 M	903	930	19 k	0	16 M
AC5TC-Recomp	3 874	0.8	2.4 M	3 519	3 357	98 M	0	305 M
STR2	483	0.7	614 k	439	455	22 M	0	0
MDD	**110**	12.4	614 k	100	100	0	0	12 M

Table 1: Results of the propagators on fully random instance set (times in seconds)

Table 1 summarizes the results, which remain similar for other parameter settings. The standard STR2 and MDD algorithms outperform our value-based propagators. Observe the large number of validity checks of AC5TC-Recomp and Q-validity checks of

AC5TC-Bool and AC5TC-CutOff, as well as the number of times they follow a pointer. AC5TC-Tr, the best value-based propagator, follows far less pointers than our other propagators because it does not follow pointers to a previously inspected tuple. Due to the lack of structure of the constraint set, the first three AC5TC propagators check multiple times the same tuples. Also, those random instances have large tables, which makes the cost of the trailable *nextTr* structure in AC5TC-Tr too high.

The Geometric Problem Instances of the geometric problem are random instances generated following a specific structure proposed by Rick Wallace [21]. 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 like in fully random CSP instances. We use the instance set from [9] 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	postT	nProp	%best	μ%best	%solv	valChk	QvalChk	pFollow
GAC3-Allowed	10.1	0.3	288 k	128	138	86	28 k	0	28 k
AC5TC-Bool	12.5	0.3	867 k	158	159	84	300	25 k	50 k
AC5TC-CutOff	10.8	0.2	867 k	137	131	86	300	25 k	50 k
AC5TC-Tr	9.6	0.8	867 k	122	200	87	300	0	13 k
AC5TC-Recomp	**7.9**	0.2	831 k	100	100	87	6 k	0	29 k
STR2	24.9	0.3	288 k	315	316	82	26 k	0	0
MDD	14.7	1.6	288 k	186	337	86	0	0	65 k

Table 2: Results of the propagators on the geom instances (times in seconds)

Table 2 presents the experimental results. The quantities are computed on instances for which none of the techniques timeout. All our propagators outperform the state-of-the-art STR and MDD. AC5TC-Tr and AC5TC-Recomp are also better than the classical AC3-Allowed. AC5TC-Recomp is the fastest on these instances. which are relatively easy and contain only binary tables. Checking the validity (not costly for binary tables) allows AC5TC-Recomp to follow less pointers than AC5TC-Bool and AC5TC-CutOff by performing longer jumps in the table. The cost of the data structures in AC5TC-Tr is too expensive and outweights its benefits. The large difference between *%best* and *μ%best* for AC5TC-Tr is due to the easiest instances, where the propagator is even more disadvantaged due to the cost of its data structures. AC5TC-Recomp also performs less validity checks than STR2 and the number of pointers followed by our propagators are less than those of MDD.

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. Those problems are modeled with binary (positive) table constraints only. 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 respectively be found in Table 3.

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
	$k = 2$							
GAC3-Allowed	16.3	0.6	1 M	173	172	166 k	0	166 k
AC5TC-Bool	18.6	0.8	2 M	197	182	576	178 k	316 k
AC5TC-CutOff	16.8	0.5	2 M	178	147	576	178 k	316 k
AC5TC-Tr	**9.4**	2.5	2 M	100	260	576	0	42 k
AC5TC-Recomp	10.1	0.4	2 M	107	106	27 k	0	154 k
STR2	26.7	1.3	1 M	283	342	46 k	0	0
MDD	26.6	3.7	1 M	282	517	0	0	307 k
	$k = 3$							
GAC3-Allowed	2.5	0.3	75 k	162	148	12 k	0	12 k
AC5TC-Bool	3.5	0.3	242 k	227	184	380	10 k	21 k
AC5TC-CutOff	2.5	0.2	242 k	163	147	380	10 k	21 k
AC5TC-Tr	2.2	0.9	242 k	140	198	380	0	4 k
AC5TC-Recomp	**1.5**	0.2	239 k	100	107	2 k	0	12 k
STR2	3.7	0.6	75 k	240	243	5 k	0	0
MDD	3.9	1.5	75 k	249	360	0	0	22 k
	$k = 4$							
GAC3-Allowed	23.4	1.3	419 k	137	155	19 k	0	19 k
AC5TC-Bool	42.5	1.6	1.6 M	250	215	677	20 k	36 k
AC5TC-CutOff	29.8	1.0	1.6 M	175	157	677	20 k	36 k
AC5TC-Tr	21.8	5.0	1.6 M	128	254	677	0	5 k
AC5TC-Recomp	**17.0**	0.8	1.58 M	100	100	3 k	0	18 k
STR2	33.2	3.3	419 k	195	277	10 k	0	0
MDD	31.2	7.3	419 k	183	392	0	0	35 k

Table 3: Experimental Results on Langford instances (times in seconds)

Except for AC5TC-Bool on the $k = 4$ set of instances, all our propagators improve the state-of-the-art STR and MDD. AC5TC-Tr and AC5TC-Recomp are also better than the classical AC3-Allowed. AC5TC-Tr is the fastest propagator for $k = 2$ and AC5TC-Recomp is the fastest on the other instance sets. Observe that the number of followed pointers is globally higher for the first instance set ($k = 2$), 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. For the last two sets, AC5TC-Tr is the second fastest propagator in terms of mean solving time. AC5TC-Tr is closer to AC5TC-Recomp on the $k = 4$ instance set. The $k = 4$ instance set includes an instance for which $n = 11$ (for $k = 3$: $n \leq 10$). Here again, the large difference between *%best* and *μ%best* for AC5TC-Tr can be attributed to the easiest instances.

Traveling Salesman Problems We conclude 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* [9]. 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, 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.

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	797	1.7	6.7 M	733	587	11 M	0	11 M
AC5TC-Bool	186	0.8	21.2 M	171	187	2 k	1 M	2 M
AC5TC-CutOff	153	0.5	21.2 M	141	144	2 k	1 M	2 M
AC5TC-Tr	120	3.3	21.2 M	111	164	2 k	0	466 k
AC5TC-Recomp	**109**	0.3	20.9 M	100	104	391 k	0	1 M
STR2	398	1.4	6.7 M	366	353	803 k	0	0
MDD	456	19.0	6.7 M	419	769	0	0	7 M

Table 4: Results of the propagators for instance set TSP-20 (times in seconds)

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	6 607	2.4	73 M	606	509	23 M	0	23 M
AC5TC-Bool	2 625	1.3	198 M	241	233	2 k	11 M	19 M
AC5TC-CutOff	1 937	0.7	198 M	178	175	2 k	11 M	19 M
AC5TC-Tr	**1 089**	5.2	198 M	100	100	2 k	0	3 M
AC5TC-Recomp	1 315	0.5	196 M	121	120	3 M	0	10 M
STR2	3 740	2.9	73 M	343	333	5 M	0	0
MDD	4 974	25.2	73 M	457	425	0	0	28 M

Table 5: Results of the propagators for instance set TSP-25 (times in seconds)

Tables 4 and 5 present the results. We first observe that STR2 and MDD perform worse than our propagators. AC5TC-Recomp is the winning strategy on *tsp-20* instances while

AC5TC-Tr is faster on the *tsp-25* ones. The latter instances are more difficult. 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-CutOff. Moreover, on these instances, the small arity makes the validity check ($O(r)$) cheap compared to Q-validity. Again, on those instances, the light-weight trailable structures of AC5TC-CutOff make it faster than AC5TC-Bool.

When we merge binary tables in tsp-20 instances into higher arity tables, we observed that AC5TC-Tr, our optimal algorithm, solves more instances than STR2, and with a smaller total execution time on the instances solved by both solvers. MDD does not compete on these instances. On simple instances, STR2 is more efficient than AC5TC-Recomp which is also more efficient than AC5TC-Tr. However, on hard instances, the optimality of AC5TC-Tr pays off and it becomes the best algorithm.

Summary We conclude that, for the fully random instances, the lack of structure in the tables prevents our propagators from competing with state-of-the-art algorithms. However, for structured instances, our propagators are faster. Globally, AC5TC-Bool and AC5TC-CutOff are slower than AC5TC-Recomp since they are testing Q-validity, not validity, and hence they perform smaller jumps in the table. Moreover, maintaining their data structures is costly. Only the optimal AC5TC-Tr outperforms AC5TC-Recomp on difficult instances while using Q-validity. However, on easier instances, the cost of its trailable *nextTr* data-structure makes it slower than AC5TC-Recomp.

7 Conclusion

This paper proposed four 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. AC5TC-Tr and AC5TC-Recomp are the two best value-based algorithms: AC5TC-Recomp does not maintain any validity information and recomputes it on demand and AC5TC-Tr 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. Our other algorithms have a time complexity of $O(r^2 \cdot t + r \cdot d)$ per table constraint. Experimental results show that on, purely random tables, our algorithms do not compete with the state-of-the-art STR2+ and MDD^c algorithms. On structured instances, our propagators outperform STR2+ and MDD^c, with a speed up varying between 1.83 and 4.57. 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 [17].

Acknowledgments The authors want to thank the anonymous reviewers for their helpful comments. 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.

References

1. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.
2. M. Carlsson. Filtering for the case constraint. Talk given at the advanced school on global constraints, 2006.
3. K. Cheng and R. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15:265–304, 2010.
4. Y. Deville and P. Van Hentenryck. Domain consistency with forbidden values. In D. Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.
5. P. J. Fleming and J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, Mar. 1986.
6. I. P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Proceedings of CP2006*, pages 182–197. Springer-Verlag, 2006.
7. I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty Second Conference on Artificial Intelligence*, pages 191–197. AAAI Press, 2007.
8. G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, pages 379–393. Springer-Verlag, 2007.
9. C. Lecoutre. *Instances of the Constraint Solver Competition*. http://www.cril.fr/∼lecoutre/.
10. C. Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley, 2009.
11. C. Lecoutre. Str2: optimized simple tabular reduction for table constraints. *Constraints*, 16:341–371, 2011.
12. C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP 06*, pages 284–298, 2006.
13. O. Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR*, pages 209–224, 2004.
14. O. Lhomme and J.-C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the Nationnal Conference on Artificial Intelligence*, pages 405–410. AAAI Press, 2005.
15. L. Perron and V. Furnon. or-tools. http:// code.google.com/p/or-tools.
16. J.-C. Régin. Improving the expressiveness of table constraints. In *Proceedings of workshop ModRef 11 at CP 11*, 2011.
17. J.-C. Régin. Improving the expressiveness of table constraints. In *In proceedings of ModRef'11 Workshop held with CP'11*, 2011.
18. J. R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007.
19. P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3):291–321, 1992.
20. P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{lin}$). *ACM Transactions on Programming Languages and Systems*, 17(4):635–671, July 1995.
21. R. Wallace. Factor analytic studies of csp heuristics. In P. van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709, pages 712–726. Springer Berlin / Heidelberg, 2005.
22. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artif. Intell.*, 171(8-9):514–534, 2007.