Extending STR to a Higher-Order Consistency

Christophe Lecoutre* and Anastasia Paparrizou† and Kostas Stergiou
CRIL-CNRS UMR 8188
Université d’Artois
F-62307 Lens, France

Dept. of Informatics and Telecommunications Engineering
University of Western Macedonia
Greece

Abstract
One of the most widely studied classes of constraints in constraint programming (CP) is that of table constraints. Numerous specialized filtering algorithms, enforcing the well-known property called generalized arc consistency (GAC), have been developed for such constraints. Among the most successful GAC algorithms for table constraints, we find variants of simple tabular reduction (STR), like STR2. In this paper, we propose an extension of STR-based algorithms that achieves full pairwise consistency (FPWC), a consistency stronger than GAC and max restricted pairwise consistency (maxRPWC). Our approach involves counting the number of occurrences of specific combinations of values in constraint intersections. Importantly, the worst-case time complexity of one call to the basic filtering procedure at the heart of our new algorithm is quite close to that of STR algorithms. Experiments demonstrate that our method can outperform STR2 in many classes of problems, being significantly faster in some cases. Also, it is clearly superior to maxRPWC+, an algorithm that has been recently proposed.

Introduction
Table constraints, i.e., constraints given in extension by listing the tuples of values allowed or forbidden by a set of variables, are widely studied in constraint programming (CP). This is because such constraints are present in many real-world applications from areas such as design and configuration, databases, and preferences’ modeling. So far, research on table constraints has mainly focused on the development of fast algorithms to enforce generalized arc consistency (GAC) which is a first-order consistency, i.e. a consistency that allows us to identify only inconsistent values (nogoods of size 1). GAC algorithms delete values from variable domains and achieve the maximum level of filtering when constraints are treated independently.

GAC algorithms for table constraints have attracted considerable interest, dating back to GAC-Schema (Bessiere and Régin 1997). Classical algorithms iterate over lists of tuples in different ways: e.g., see (Bessiere and Régin 1997; Lhomme and Régin 2005; Lecoutre and Szymanek 2006). Recent developments, however, suggested maintaining dynamically the list of supports in constraint tables: these are the variants of simple tabular reduction (STR) (Ullmann 2007; Lecoutre 2011; Lecoutre, Likitvivatanavong, and Yap 2012). Alternatively, specially-constructed intermediate structures such as tries (Gent et al. 2007) or multi-valued decision diagrams (MDDs) (Cheng and Yap 2010) have been proposed. A more recent development of AC5-based algorithms has also been proposed in (Mairy, Van Hentenryck, and Deville 2012), but its relevance has been shown on binary/ternary constraints only. Among this variety of algorithms, STR2 along with the MDD approach are considered to be the most efficient ones (especially, for large arity constraints).

A different line of research has investigated stronger consistencies and algorithms to enforce them. Some of them are first-order consistencies, e.g., see (Debruyne and Bessiere 2001; Bessiere, Stergiou, and Walsh 2008; Karakashian et al. 2010) whereas a few other ones are higher-order, e.g., see (Montanari 1974; Janssen et al. 1989; Jégou 1993; Lecoutre, Cardon, and Vion 2011) indicating that inconsistent tuples of values (nogoods of size 2 or more) can be identified. In contrast to GAC algorithms, the proposed algorithms to enforce these stronger and/or higher-order consistencies are able to reason from several constraints simultaneously, as, for example, constraint intersections with pairwise consistency (PWC) (Janssen et al. 1989). Most of these methods are generic, since they are applicable on constraints of any type, which typically results in a high computation cost. A specialized algorithm for table constraints that achieves a consistency stronger than GAC was proposed very recently (Paparrizou and Stergiou 2012). This algorithm, called maxRPWC+, extends the GAC algorithm of (Lecoutre and Szymanek 2006) and enforces a domain-filtering restriction of PWC, called max restricted pairwise consistency (maxRPWC) (Bessiere, Stergiou, and Walsh 2008). Interestingly, it achieves good performance on several classes of problems, but it was not tested against state-of-the-art GAC algorithms.

In this paper, we propose a new higher-order consistency algorithm for table constraints, called FPWC-STR, based on
STR. Actually, we show that all STR-based algorithms can be easily extended to achieve stronger pruning by introducing a set of counters for each intersection between any two constraints $c_i$ and $c_j$. At any time each counter in this set holds the number of valid tuples in $c_i$’s table that include a specific combination of values for the set of variables that are common to both $c_i$ and $c_j$. We show that $FPWC-STR$ enforces full pairwise consistency, i.e., both PWC and GAC, and we prove that it also guarantees maxRPWC. Importantly, the worst-case time complexity of one call to the basic filtering procedure at the heart of $FPWC-STR$ is quite close to that of STR algorithms. Our experiments demonstrate that $FPWC-STR$ can outperform STR2 on many classes of problems with intersecting table constraints (being significantly faster in some cases), and is also typically considerably faster than maxRPWC+ (often by very large margins).

Background

An instance of the constraint satisfaction problem (CSP) is defined by a constraint network (CN) which is a triplet $(X, D, C)$ where $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains, one for each variable, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. We will denote by $d$ the maximum cardinality of domains. For simplicity, a pair $(x, a)$ such that $x \in X$ and $a \in D(x)$ is called a value (of the CN).

A positive table constraint is a constraint given in extension and defined by a set of allowed tuples. Each table constraint $c_i$ is a pair $(scp(c_i), \text{table}(c_i))$, where $scp(c_i) = \{x_{i_1}, \ldots, x_{i_k}\}$ is an ordered subset of $X$ referred to as the constraint scope, and $\text{table}(c_i)$ is a subset of the Cartesian product $D(x_{i_1}) \times \ldots \times D(x_{i_k})$ that specifies the allowed combinations of values (tuples) for the variables in $scp(c_i)$; $\tau$ denotes the arity of $c_i$. Each tuple $\tau \in \text{table}(c_i)$ is an ordered list of values. Given a (table) constraint $c_i$ and a tuple $\tau \in \text{table}(c_i)$, we denote by $\tau[x]$ the projection of $\tau$ on a variable $x \in scp(c_i)$ and by $\tau[X]$ the projection of $\tau$ on any subset $X \subseteq scp(c_i)$ of variables; $\tau[X]$ is called a subtuple of $\tau$. A tuple is valid iff none of the values in the tuple has been removed from the variable domains. Given two constraints $c_i$ and $c_j$, if $|scp(c_i) \cap scp(c_j)| > 1$ we say that the constraints intersect non-trivially.

The most commonly used local consistency in constraint solvers is generalized arc consistency (GAC). A value $(x, a)$ is GAC iff $\forall c \in C \ | \ x \in scp(c) \rightarrow \exists \tau \in \text{table}(c) \text{ such that } \tau[x] = a$ and $\tau$ is valid. In this case $\tau$ is called a support of $(x, a)$ on $c$. A constraint $c$ is GAC iff all values $(x, a)$ with $x \in scp(c)$ and $a \in D(x)$ are GAC. A CN is GAC iff all its values are GAC. Among local consistencies stronger than GAC, max restricted pairwise consistency (maxRPWC) looks quite promising (Bessiere, Stergiou, and Walsh 2008). A value $(x, a)$ is maxRPWC iff $\forall c_i \in C \ | \ x \in scp(c_i), (x, a)$ has a support $\tau_i$ on $c_i$ such that $\forall c_j \in C, \exists \tau_j \in \text{table}(c_j) \ | \ \tau_j[scp(c_i) \cap scp(c_j)] = \tau_i[scp(c_i) \cap scp(c_j)]$ and $\tau_j$ is valid. In this case, we say that $\tau_j$ is a PW-support of $\tau_i$ in $\text{table}(c_j)$ and $\tau_i$ is a maxRPWC-support of $(x, a)$. A CN is maxRPWC iff all its values are maxRPWC.

GAC and maxRPWC are first-order consistencies, thereby filtering variable domains. Higher-order consistencies allow us to identify inconsistent tuples of values. This is the case of pairwise consistency (PWC) defined as follows (for positive table constraints). A tuple $\tau_i$ in the table of a constraint $c_i$ is PWC iff $\forall c_j \in C, \exists \tau_j$ in $\text{table}(c_j)$ which is a PW-support of $\tau_i$. A CN is PWC iff every tuple of every constraint of $P$ is PWC. We will also say that a CN is PWC+GAC (resp, PWC+maxRPWC) iff it is both PWC and GAC (resp, PWC and maxRPWC). Finally, full pairwise consistency (FPWC) is the name we choose for PWC+GAC.

Following (Debruyne and Bessiere 2001), a local consistency $\phi$ is stronger than $\psi$ iff in any CN in which $\phi$ holds then $\psi$ holds, and strictly stronger iff it is stronger and there is one CN for which $\psi$ holds but $\phi$ does not. Accordingly, $\phi$ is incomparable to $\psi$ iff neither is stronger than the other.

Extending STR

In this section, we present a simple way to filter domains and constraints by using the technique of simple tabular reduction (STR), together with a few additional data structures related to (sub)tuple counting. We explain why the update/restoration and the exploitation of introduced counters is interleaved with STR in a seamless way to obtain specialized and efficient higher-order consistency algorithms for table constraints. The new algorithms we propose will be called eSTR*, derived from extended STR. The ‘*’ stands for a particular STR algorithm (e.g., when extending STR2 we name the algorithm eSTR2).

The central idea of eSTR* is to store the number of times that each subtuple appears in the intersection of any two constraints. Specifically, for each constraint $c_i$, we introduce a set of counters for each (non trivial) intersection between $c_i$ and another constraint $c_j$. Assuming that $Y$ is the set of variables that are common to both $c_i$ and $c_j$, at any time each counter in this set holds the number of valid tuples in $c_i$’s table that include a specific combination of values for $Y$. In this way, once a tuple $\tau \in \text{table}(c_j)$ has been verified as valid, we can check if it has a PW-support in $\text{table}(c_i)$ simply by observing the value of the corresponding counter (i.e., the counter for subtuple $\tau[scp(c_j) \cap scp(c_i)]$). If this counter is greater than 0 then $\tau$ has a PW-support in $\text{table}(c_i)$. Importantly, this check is done in constant time.

Note that our approach is related to that in (Samaras and Stergiou 2005), where arc consistency is enforced on the dual representation of non-binary problems using counters that record information about constraint intersections. However, for any two constraints that intersect, the space complexity of that approach is exponential in the size of the subset of variables belonging to the intersection. Counters have also been exploited in algorithms AC4/GAC4 (Mohr and Henderson 1986; Mohr and Masini 1988). Finally, there exist some connections with both the MDD-based propagation approach (Hoda, van Hoeve, and Hooker 2010), because in valid tuples are aimed at being removed, and the intersection encoding of sliding constraints (Bessiere et al. 2008).

Algorithm 1 presents the main framework for eSTR* by extending the basic STR algorithm, as proposed in (Ullmann 2007). We choose to present an extension of STR, simply called eSTR, because of its simplicity compared to STR2 and STR3, which can be extended in a very similar way.
Here, we consider a constraint-based version of STR, meaning that the propagation queue, denoted by $Q$, handles constraints, because it is quite adapted to our filtering operations. The level of local consistency achieved by means of the process of propagation will be discussed in the next section. Whenever a constraint is removed from the queue, STR iterates over the valid tuples in the constraint and removes any tuple that has become invalid through the deletion of one of its values (the while loop in Algorithm 1; see lines 4–14). Thus, only valid tuples are kept in tables. After finishing the iteration, all values that are no longer supported are deleted (the for loop in Algorithm 1; see lines 15–21) and for each variable $x$ whose domain has been reduced, all constraints involving $x$ are added to the propagation queue $Q$, excluding the currently processed constraint (lines 20–21).

For each (positive table) constraint $c$ of the CN, we have the following STR data structures:

- $\text{table}[c]$ is the set of allowed tuples associated with $c$. This set is represented by an array of tuples indexed from 1 to $\text{table}[c].\text{length}$ which denotes the size of the table.
- $\text{position}[c]$ is an array of size $\text{table}[c].\text{length}$ that provides direct access to the tuples of $c$. At any given time the values in $\text{position}[c]$ are a permutation of $\{1, 2, \ldots, t\}$, where $t$ is the size of the table. The $i$th tuple of $c$ is $\text{table}[c][\text{position}[c][i]]$.
- $\text{currentLimit}[c]$ is the position of the last valid tuple in $\text{table}[c]$. The current table of $c$ is composed of exactly $\text{currentLimit}[c]$ tuples. The values in $\text{position}[c]$ at indices ranging from 1 to $\text{currentLimit}[c]$ are the positions of the currently valid tuples of $c$.
- $\text{pwValues}[c][x]$ is a set that contains for variable $x$ in $\text{scp}(c)$ all values in $D(x)$ proved to have a maxRPWC-support when $\text{eSTR}$ is applied on $c$. This structure is quite similar to that of STR, called $\text{gacValues}[x]$.

Structures $\text{position}[c]$ and $\text{currentLimit}[c]$, which basically implement the structure called sparse set (Briggs and Torczon 1993), allow restoration of deleted tuples in constant time (during backtrack search); for more information, see (Ullmann 2007). We now describe, for each non trivial intersection of a constraint $c$ with a constraint $c_i$, the additional structures used in $\text{eSTR}$:

- $\text{ctr}[c][c_i]$ is an array that stores the counters associated with the intersection of $c$ with $c_i$. For each subtuple for variables in $\text{scp}(c) \cap \text{scp}(c_i)$ that appears in at least one tuple of $\text{table}(c)$, there is a counter $\text{ctr}[c][c_i][j]$ that holds the number of valid tuples in $\text{table}[c]$ that include that subtuple. The value of the index $j$ can be found from any tuple in $\text{table}[c]$ using the next structure.
- $\text{ctrIndexes}[c][c_i]$ is a set of indexes for the tuples of $\text{table}[c]$. For each tuple $\tau$, this data structure holds the index of the counter in $\text{ctr}[c][c_i]$ that is associated with the subtuple $\tau(\text{scp}(c) \cap \text{scp}(c_i))$. $\text{ctrIndexes}[c][c_i]$ is implemented as an array of size $\text{table}[c].\text{length}$.

$\text{ctrLink}[c][c_i]$ is an array of size $\text{ctr}[c][c_i].\text{length}$ that links $\text{ctr}[c][c_i]$ with $\text{ctr}[c][c_i]$. For each counter $\text{ctr}[c][c_i][j]$ corresponding to a subtuple for variables in $\text{scp}(c) \cap \text{scp}(c_i)$, $\text{ctrLink}[c][c_i][j]$ holds the index of the counter in $\text{ctr}[c][c_i]$ that is associated with that subtuple. If the subtuple is not included in any tuple of $\text{table}[c]$ then $\text{ctrLink}[c][c_i][j]$ is set to NULL.

Figure 1 illustrates $\text{eSTR}$’s data structures. There are two constraints $c_1$ and $c_2$ intersecting on variables $x_2$ and $x_3$. Three different subtuples for variables $x_2$ and $x_3$ are present in $\text{table}[c_1]$: $(0, 0), (0, 1)$ and $(1, 0)$. Hence, there are three counters in $\text{ctr}[c_1][c_2]$. Each counts the number of times a specific subtuple appears in $\text{table}[c_1]$. For each tuple in $\text{table}[c_1]$, the corresponding entry in $\text{ctrIndexes}[c_1][c_2]$ gives the index of the counter in $\text{ctr}[c_1][c_2]$ associated with the underlying subtuple. For each counter in $\text{ctr}[c_1][c_2]$, the corresponding entry in $\text{ctrLink}[c_1][c_2]$ gives the index of the counter in $\text{ctr}[c_2][c_1]$ associated with the same subtuple. Since subtuple $(0, 1)$ does not appear in $\text{table}[c_2]$, the entry in $\text{ctrLink}[c_1][c_2]$ for this subtuple is NULL.

The behaviour of $\text{eSTR}$ is identical to that of STR, except: 1) it applies an extra check for PWC when a tuple is verified as valid, and 2) it decrements (resp. increments) the corresponding counters when a tuple is removed (resp. restored). Also, $\text{eSTR}$ needs to build its data structures in an initialization step. This is done by traversing each $\text{table}[c]$ exactly once. At the end of this step all counters are set to their proper values.

We now describe the auxiliary functions used by the main algorithm, with a special emphasis on those that are specific to $\text{eSTR}$. Function $\text{isValidTuple}$ takes a tuple $\tau$ and returns true iff $\tau$ is valid. Function $\text{removeTuple}$ takes a tuple $\tau$ and removes it in constant time by replacing $\text{position}[\tau][i]$, where $i$ is the position of $\tau$ in $\text{table}[c]$, with $\text{position}[\tau][\text{currentLimit}[c]]$ (namely by swapping indexes and not tuples) and then decrementing $\text{currentLimit}[c]$ by one. Function 2, $\text{isPWconsistent}$, specific to $\text{eSTR}$ is called at line 7 of Algorithm 1 whenever a tuple $\tau \in \text{table}(c)$ has been verified as valid. This function iterates over each constraint $c_i$ that intersects with $c$ and verifies if $\tau$ has a PW-support in $\text{table}(c_i)$ or not. This is done through a look-up in the appropriate counter in constant time. Specifically, using structures $\text{ctrIndexes}[c][c_i]$ and $\text{ctrLink}[c][c_i]$ we locate the appropriate counter in $\text{ctr}[c][c_i]$ and check its value. If it is neither NULL nor 0,
then τ is PW-supported. Otherwise, FALSE is returned in order to get τ removed. Function 3, updateCtr, specific to eSTR is called at line 14 of Algorithm 1 in order to update some counters just after a tuple has been removed. For each constraint c, that intersects with c, j ← ctrIndexes[𝑐][index] is located. The variable j represents the index for the subtuple of the removed tuple τ in the array of counters concerning the intersection of c with c. Then the corresponding counter in ctr[c][j] can be decremented. If the value of this counter becomes 0 then this means that some tuples in table[c] have lost their last PW-support in table[c]. Since this may cause value deletions for the variables in scp(c), constraint c is added to Q so that it can be processed again.

Algorithm 1 eSTR(c; constraint)
1: for each unassigned variable x ∈ scp(c) do
2: pvValues[x] ← ∅
3: i ← 1
4: while i ≤ currentLimit[c] do
5: index ← position[c][i]
6: τ ← table[c][index]
7: if isValidTuple(τ, c) AND isPWconsistent(c, index) then
8: for each unassigned variable x ∈ scp(c) do
9: if τ[x] ∈ pvValues[x] then
10: pvValues[x] ← pvValues[x] \ τ[x]
11: i ← i + 1
12: else
13: removeTuple(τ, c) // currentLimit[c] decremented
14: updateCtr(τ, c) // Counters in ctr[c] decremented
// domains are updated and constraints are enqueued
15: for each unassigned variable x ∈ scp(c) do
16: if pvValues[x] ⊆ D(x) then
17: D(x) ← pvValues[x]
18: if D(x) = ∅ then
19: return FAIL
20: for each constraint c’ such that c’ ⊆ c x ∈ scp(c’) do
21: add c’ to Q
22: return SUCCESS

Function 2 isPWconsistent(c, index): Boolean
1: for each c, i ∈ scp(c) \ scp(c) > 1 do
2: j ← ctrIndexes[𝑐][index]
3: k ← ctrLink[𝑐][j]
4: if k = NULL OR ctr[c][k] = 0 then
5: return FALSE
6: return TRUE

Function 3 updateCtr(τ, c)
1: for each c, i ∈ var(𝑐) \ var(𝑐) > 1 do
2: j ← ctrIndexes[𝑐][index]
3: ctr[c][j] ← ctr[c][j] − 1
4: if ctr[c][j] = 0 then
5: add c to Q

Finally, when a failure occurs in the context of a backtrack search, certain values must be restored to domains. Consequently, tuples that were invalid may now become valid and thus must be restored. For each constraint c this is achieved in constant time by STR by just updating currentLimit[c]. In addition, eSTR updates all the affected counters by iterating through all tuples being restored and incrementing the corresponding counters for every cj that intersects with c (i.e. ctr[c][cj]). This costs O(gt) in the worst case, where t the size of c and g the number of constraints intersecting with c. However, it is much faster in practice since usually only a few tuples are restored after each failure. Note that currentLimit[c] allows us to easily locate restored tuples.

Enforcing FPWC
Assuming a CN P only involving positive table constraints, Algorithm 4, FPWC-STR, shows the full process of propagating constraints of P by calling procedure eSTR iteratively through the use of a propagation queue Q. Recall that Q may be updated when calling eSTR on a constraint c at lines 20–21 of Algorithm 1 and also at line 5 of Function 3. A weak version of FPWC-STR, denoted by FPWC-STRw can be obtained by discarding lines 4–5 of Function 3 (i.e., the update of Q is ignored when a PW-support is lost).

Algorithm 4 FPWC-STR(P = (𝑋; ∅; C) : CN)
1: Q ← c
2: while Q ≠ ∅ do
3: pick and delete c from Q
4: if eSTR(c) = FAIL then
5: return FAIL
6: return SUCCESS

Proposition 1 Algorithm FPWC-STR applied to a CN P enforces full pairwise consistency on P.

Proof: Clearly FPWC-STR enforces GAC because each call of the form eSTR(c) guarantees that c is made GAC and everytime a value is deleted for a variable x, all constraints involving x are enqueued (and also, all constraints are enqueued initially). Now, let us consider a tuple τ, in the table of a constraint c, which is not PW. This means (by definition of PW) that there exists a constraint c, non trivially intersecting c such that no PW-support of τ in table[c] exists. Because everytime a tuple is deleted, the counters of underlying subtuples corresponding to constraint intersections are updated (decremented), and also considering the way these counters are initialized, during the execution of the algorithm FPWC-STR, we will necessarily have ctr[c][k][|k] set to value NULL or 0 where k is the index for the subtuple τ[scp(c) \ scp(c)] in this array of counters. Besides, the constraint c will necessarily be processed after ctr[c][k][|k] reaches 0 (resp., after it is initialized to NULL) because of the execution of lines 4-5 of function updateCtr that adds c to Q (resp., because c is put in Q initially). When c is processed, the tuple τ will be deleted because isPWconsistent will return FALSE. Consequently, any tuple that is not PW is deleted by our algorithm FPWC-STR. We can conclude that FPWC-STR enforces both PWC+GAC, i.e., full pairwise consistency.

It is interesting to note that FPWC guarantees maxRPWC as shown by the following proposition.

Proposition 2 PWC+GAC and PWC+maxRPWC are equivalent

Proof: On the one hand, clearly PWC+maxRPWC is stronger than PWC+GAC since maxRPWC is stronger than GAC. On the other hand, let us assume a CN P which is PW and a value (x, a) of P which is not maxRPWC. This means (by definition) that there exists a constraint c involving x such that either (x, a) has no support on c, or (x, a) has no maxRPWC-support on c. However, because
If $P$ is PWC, the reason why $(x, a)$ is not maxRPWC is necessarily that $(x, a)$ has no support on $c$. In other words, $(x, a)$ is not GAC. We deduce that PWC+GAC is stronger than PWC+maxRPWC, and finally that PWC+GAC and PWC+maxRPWC are equivalent.

**Proposition 3** The consistency level achieved by Algorithm FPWC-STR$^w$ is incomparable to maxRPWC and PWC.

The proof is omitted (due to lack of space). We now analyze the worst-case time and space complexities of eSTR, the basic filtering procedure associated with each table constraint in FPWC-STR.

**Proposition 4** The worst-case time complexity$^2$ of one call to eSTR is $O(rd + max(r, g)t)$ where $r$ denotes the arity of the constraint, $t$ the size of its table and $g$ the number of intersecting constraints.

**Proof:** Recall that the worst-case time complexity of STR is $O(rd + rt)$ (Lecoultre 2011). The application of eSTR on a constraint $c$ is identical to that of STR except for the calls to isPWconsistent and updateCtr in lines 7 and 14 of Algorithm 1, respectively. In both functions, the algorithm iterates over the set of $g$ constraints intersecting with $c$, and for each one performs a constant time operation. Hence, the complexity of eSTR is $O(rd + max(r, g)t))$.

One may be surprised by the fact that the worst-case time complexity of eSTR is close to that of STR, although a stronger filtering is achieved. However, the difference can be emphasized when we consider the maximum number of repeated calls to the function eSTR for a given constraint $c$. For STR, this is $O(rd)$ because after each removal of a value (for a variable in the scope of $c$), one call eSTR($c$) is possible. For eSTR, this is $O(max(rd, t))$ because one call is possible after each value deletion but also after each loss of a PW-support for a tuple in $table(c)$. Note that when we consider FPWC-STR$^w$, for eSTR we have $O(rd)$ as for STR: this is the reason why we introduce this variant. Overall, our intuition is that for many problems, the number of repeated calls to the filtering procedure of the same constraint is limited.

**Proposition 5** The worst-case space complexity of eSTR for handling one constraint is $O(n + max(r, g)t)$.

**Proof:** Recall that the worst-case space complexity of STR is $O(n + rt)$ per constraint (Lecoultre 2011). Each additional eSTR structure is $O(t)$ per intersecting constraint, giving $O(gt)$.

It is possible to reduce the memory requirements in two ways. First, by replacing the at most $eg$ sets of counters with $e$ sets, one for each constraint, in order to reduce the size of $ctr$ and $ctrLink$. Second, by using a hash function to map each tuple $\tau \in table(c)$ to its associated counters in $ctr[c]$. This would make the use of $ctrIndexes$ obsolete.

To emphasize the difference between FPWC-STR and FPWC-STR$^w$, let us consider the CN $P$ depicted in Figure 2. There are five variables $\{x_1, \ldots, x_5\}$ with domain $\{0, 1\}$, one variable $x_6$ with domain $\{0\}$, and three positive table constraints $c_1$, $c_2$ and $c_3$ (with their allowed tuples shown). One can check that $P$ is maxRPWC. For example, the value $(x_1, 0)$ admits $(0, 0, 0)$ as support on $c_1$ and $(0, 0, 0, 0)$ as PW-support of $(0, 0, 0)$ in $table(c_2)$. However $P$ is not PWC. Indeed, the tuple $(0, 0, 0, 0)$ in $table(c_2)$ has no PW-support in $table(c_3)$. Consequently, FPWC-STR deletes this tuple, and $(x_1, 0)$ when $c_1$ is processed. Now, with FPWC-STR$^w$, if constraint $c_1$ is processed first in our example then no value deletion can be made. This is because when $c_1$ is processed, the tuple $(0, 0, 0)$ in $table(c_1)$ admits $(0, 0, 0, 0)$ as PW-support in $table(c_2)$. When $c_2$ is later processed, the tuple $(0, 0, 0, 0)$ is removed but no value for variables in $scp(c_2)$ can be deleted. This means that the propagation queue is left unchanged. Therefore, $c_1$ will not be processed again, and value $(x_1, 0)$ will not be deleted. Hence, the pruning power of FPWC-STR$^w$ cannot be characterized precisely because it depends on the ordering of the propagation queue.

**Experimental Results**

We ran experiments on benchmark problems from the CSP Solver Competition$^3$. We tried the following classes that include table constraints with non-trivial intersections: random problems, forced random problems, aim-100 and aim-200, Dubois, positive table constraints, and BDD. We compared algorithms STR2, maxRPWC+, FPWC-STR, FPWC-STR$^w$ (for abbreviation the latter two will be called eSTR2 and eSTR$^w$ hereafter). All four were implemented within a CP solver written in Java and tested on an Intel Core i5 of 2.40GHz processor and 4GB RAM. A cpu time limit of 6 hours was set. Search used the $dom/deg$ heuristic for variable ordering and lexicographical value ordering. We chose $dom/deg$ as opposed to $dom/ddeg$ because the decisions made by the latter are influenced by the ordering of the propagation queue making it harder to objectively compare the pruning efficiency of the algorithms.

Table 1 shows the mean cpu times (in secs) obtained by the tested algorithms on each problem class for initialization and preprocessing. Also, it shows the mean cpu times and numbers of nodes obtained by backtracking algorithms that apply the propagation methods throughout search. During initialization, the data structures of an algorithm are initialized, while preprocessing includes one run of a propagation algorithm. In Table 2 we present results from selected instances focusing on the search effort. Search results from

---

$^2$We omit to consider lines 20-21 because they concern propagation (and were hidden in the description of STR).

$^3$http://www.cril.univ-artois.fr/CPAI08/
Table 1: Mean cpu times for initialization (i), preprocessing (p), search (s), and mean numbers of visited nodes (n).

<table>
<thead>
<tr>
<th>Problem Class</th>
<th>STR2</th>
<th>maxRPWC+</th>
<th>eSTR2w</th>
<th>eSTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random-fcd</td>
<td>0.02</td>
<td>0.3</td>
<td>0.67</td>
<td>0.69</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>0.1</td>
<td>0.2</td>
<td>0.09</td>
<td>0.13</td>
</tr>
<tr>
<td>Random</td>
<td>0.02</td>
<td>0.3</td>
<td>0.63</td>
<td>0.62</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>0.09</td>
<td>0.2</td>
<td>0.08</td>
<td>0.14</td>
</tr>
<tr>
<td>Positive table-8</td>
<td>0.08</td>
<td>1.8</td>
<td>76</td>
<td>85</td>
</tr>
<tr>
<td>#Inst=20</td>
<td>0.3</td>
<td>0.4</td>
<td>0.9</td>
<td>1.7</td>
</tr>
<tr>
<td>Positive table-10</td>
<td>0.06</td>
<td>0.3</td>
<td>12.2</td>
<td>16.2</td>
</tr>
<tr>
<td>#Inst=20</td>
<td>0.07</td>
<td>1.847</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>BDD</td>
<td>0.24</td>
<td>9.3</td>
<td>mem</td>
<td>mem</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>1.4</td>
<td>6.2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dubois</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.02</td>
</tr>
<tr>
<td>#Inst=8</td>
<td>0.02</td>
<td>0.002</td>
<td>0.002</td>
<td>0</td>
</tr>
<tr>
<td>Aim-100</td>
<td>0.11</td>
<td>0.29</td>
<td>0.20</td>
<td>0.21</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>0.002</td>
<td>0.04</td>
<td>0.012</td>
<td>0.003</td>
</tr>
<tr>
<td>Aim-200</td>
<td>0.39</td>
<td>0.58</td>
<td>0.32</td>
<td>0.33</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>0.004</td>
<td>0.1</td>
<td>0.01</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Aim-200 were obtained using dom/wdeg for variable ordering because this class is hard for dom/degg.

As expected, eSTR2 and its weak version typically have much higher initialization times than STR2 and are usually slower during preprocessing. They are particularly expensive on classes of problems which include intersections on large sets of variables, as is the case with the BDD and Positive-table classes. BDD instances consist of constraints with arity 18 that intersect on as many as 16 variables. In addition, the constraints are very loose. As a result, eSTR2 (and eSTR2w) exhausts all of the available memory when trying to build its data structures.

Regarding the cost of our algorithms during initialization and preprocessing compared to maxRPWC+, results vary. For example, on the Positive table classes maxRPWC+ is much faster during initialization. However, our algorithms are many orders of magnitude faster during the preprocessing of Positive table-10 instances which are usually proven unsatisfiable by these algorithms without search.

Comparing eSTR2 to eSTR2w with respect to search effort, we can make two observations: First, the extra filtering of eSTR2 does pay off on some classes as node counts are significantly reduced (Aim-100) while on other classes it does not (Random). Second, the much higher time complexity bound of eSTR2 is not really visible in practice. eSTR2w is faster than eSTR2 on average, but the differences are not very significant.

Comparing our algorithms to STR2 it seems that there are problem classes where they can be considerably more efficient. This is definitely the case with the Aim classes where eSTR2w and eSTR2 can outperform STR2 by several orders of magnitude on some instances, being one order of magnitude faster on average in the Aim-100 class. Also, there can be significant differences in favor of eSTR2w on instances of other classes, such as Random, Random-forced, and Dubois. On the other hand, if we consider the performance of the algorithms during both initialization and search, STR2 is better than the proposed methods on Positive table problems and of course BDD.

Finally, comparing our algorithms to maxRPWC+ it is clear that they are superior as they are faster on all the tested classes (except BDD). The differences in favor of eSTR2 and eSTR2w can be very large. For example in the Positive table classes they are faster by orders of magnitude.

**Conclusion**

In this paper, we have introduced a new higher-order consistency algorithm for table constraints that enforces full pair-

Table 2: Cpu times (t) in secs and nodes (n).

<table>
<thead>
<tr>
<th>Instance</th>
<th>STR2</th>
<th>maxRPWC+</th>
<th>eSTR2w</th>
<th>eSTR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand-3-20-20-60</td>
<td>130</td>
<td>102</td>
<td>37</td>
<td>66</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>n</td>
<td>128,221</td>
<td>33,924</td>
<td>27,490</td>
</tr>
<tr>
<td>rand-3-20-20-60</td>
<td>430</td>
<td>183</td>
<td>43</td>
<td>80</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>n</td>
<td>534,012</td>
<td>38,556</td>
<td>26,531</td>
</tr>
<tr>
<td>rand-3-20-20-60</td>
<td>450</td>
<td>536</td>
<td>187</td>
<td>220</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>n</td>
<td>462,920</td>
<td>129,618</td>
<td>121,199</td>
</tr>
<tr>
<td>rand-3-20-20-60</td>
<td>531</td>
<td>295</td>
<td>74</td>
<td>137</td>
</tr>
<tr>
<td>#Inst=50</td>
<td>n</td>
<td>827,513</td>
<td>64,665</td>
<td>45,268</td>
</tr>
<tr>
<td>rand-8-20-5-18</td>
<td>17</td>
<td>753</td>
<td>30</td>
<td>26</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>17,257</td>
<td>3,430</td>
<td>1,001</td>
</tr>
<tr>
<td>rand-8-20-5-18</td>
<td>19</td>
<td>1,568</td>
<td>52</td>
<td>55</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>67,803</td>
<td>7,920</td>
<td>3,299</td>
</tr>
<tr>
<td>rand-10-10-10-5</td>
<td>0.4</td>
<td>208</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>#Inst=20</td>
<td>n</td>
<td>1,110</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rand-10-10-10-5</td>
<td>0.4</td>
<td>1,687</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>#Inst=20</td>
<td>n</td>
<td>1,110</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dubois-22</td>
<td>30</td>
<td>1.5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>20,582</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>dubois-27</td>
<td>315</td>
<td>734</td>
<td>96</td>
<td>182</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>129,062,226</td>
<td>41,538,898</td>
<td>41,538,898</td>
</tr>
<tr>
<td>aim-100-1-6-sat-2</td>
<td>423</td>
<td>0.16</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>29,181,742</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>aim-100-2-0-sat-3</td>
<td>2,447</td>
<td>0.3</td>
<td>0.14</td>
<td>0.05</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>177,832,989</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>aim-200-2-0-sat-1</td>
<td>57</td>
<td>0.7</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>2,272,993</td>
<td>1,782</td>
<td>9,847</td>
</tr>
<tr>
<td>aim-200-2-0-sat-4</td>
<td>30</td>
<td>0.7</td>
<td>0.4</td>
<td>0.2</td>
</tr>
<tr>
<td>#Inst=10</td>
<td>n</td>
<td>987,160</td>
<td>1,965</td>
<td>4,276</td>
</tr>
</tbody>
</table>
wise consistency. It is based on an original combination of two techniques that have proved their worth: simple tabular reduction and tuple counting. This algorithm, and its weak variant, have been shown to be highly competitive on many problems with intersecting constraints.

References


