Strong Local Consistency Algorithms for Table Constraints *

Anastasia Paparrizou† Kostas Stergiou

Abstract

Table constraints are important in constraint programming as they are present in many real problems from areas such as configuration and databases. As a result, numerous specialized algorithms that achieve generalized arc consistency (GAC) on table constraints have been proposed. Since these algorithms achieve GAC, they operate on one constraint at a time. In this paper we propose new filtering algorithms for positive table constraints that achieve stronger local consistency properties than GAC by exploiting intersections between constraints. The first algorithm, called `maxRPWC+`, is a domain filtering algorithm that is based on the local consistency maxRPWC and extends the GAC algorithm of Lecoutre and Szymanek [23]. The second algorithm extends the state-of-the-art `STR`-based algorithms to stronger relation filtering consistencies, i.e., consistencies that can remove tuples from constraints' relations. Experimental results from benchmark problems demonstrate that the proposed algorithms are quite competitive with standard GAC algorithms like `STR2` in some classes of problems with intersecting table constraints, being orders of magnitude faster in some cases.

1 Introduction

Table constraints, i.e., constraints given in extension, are ubiquitous in constraint programming (CP). First, they naturally arise in many real applications from areas such as configuration and databases. And second, they are a useful modeling tool that can be called upon to, for instance, easily capture preferences [15]. Given their importance in CP, it is not surprising that table constraints are among the most widely studied constraints and as a result numerous specialized algorithms that achieve generalized arc consistency (GAC) on them have been proposed.

GAC algorithms for positive table constraints, i.e., constraints defined by sets of allowed tuples, have received the bulk of the attention. Such algorithms utilize a number of different techniques to speed-up the check for generalized arc consistency. For example, some of them build upon GAC-schema [4] by interleaving the exploration of allowed and valid tuples using either intricate data structures [26] or binary search [23]. Other methods compile the tables into efficient data structures that allow for

*Some results included in this paper first appeared in [31].

†This author has been funded by the EU project ICON (FP7-284715).

faster support search (e.g., [11] and [9]). Simple Tabular Reduction (STR) [34] and its refinements [18, 22] maintain dynamically the support tables by removing invalid tuples from them during search. A recent approach holds information about removed values in the propagation queue and utilizes it to speed up support search [29].

Given that GAC is a property defined on individual constraints, algorithms for GAC operate on one constraint at a time trying to filter infeasible values from the variables of the constraint. A different line of research has investigated stronger consistencies and algorithms to enforce them. Some of them are domain filtering, meaning that they only prune values from the domains of variables, e.g., see [10, 7], whereas a few other ones are higher-order (or relation filtering), e.g., see [12, 13, 35, 16, 20], 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 consistencies are able to consider several constraints simultaneously. For example, pairwise consistency (PWC) [12] is a relation filtering consistency that considers intersections between pairs of constraints.

Recently there has been renewed interest for strong domain or relation filtering local consistencies as new ones have been proposed or/and efficient algorithms for existing ones have been devised [19, 7, 16, 37]. One of the most promising such consistencies is Max Restricted Pairwise Consistency (maxRPWC) [7], which is local consistency that is based PWC but can only make value deletions. In practice, strong consistencies are mainly applicable on constraints that are extensionally defined since intensionally defined constraints usually have specific semantics and are provided with efficient specialized filtering algorithms. However, a significant shortcoming of existing works on maxRPWC and other strong local consistencies is that the proposed algorithms for them are generic. That is, like earlier GAC algorithms such as GAC2001/3.1 [6], they are designed to operate on both intensional and extensional constraints, failing to recognize that strong consistencies are predominantly applicable on extensional constraints and should thus focus on such constraints.

Despite the wealth of research on strong consistencies, they have not been widely adopted by CP solvers. State-of-the art solvers such as Gecode, Abscon, Choco, Minion, etc. predominantly apply GAC, and lesser forms of consistency such as bounds consistency, when propagating constraints. Regarding table constraints, CP solvers typically offer one or more of the above mentioned GAC methods for propagation.

In this paper we propose filtering algorithms that achieve stronger consistency properties than GAC and have been specifically designed for table constraints. We contribute to both directions of domain and relation filtering methods by extending existing GAC algorithms for table constraints. The proposed methods are a step towards the efficient handling of intersecting table constraints and also provide specialization of strong local consistencies to extensional constraints that can be useful in practice.

The first algorithm, called maxRPWC+, extends the GAC algorithm for table constraints of Lecoutre and Szymanek [23] and specializes the generic maxRWPC algorithm maxRPWC1 [7]. The proposed domain filtering algorithm incorporates several techniques that help alleviate redundancies (i.e., redundant constraint checks and other operations on data structures) displayed by existing maxRPWC algorithms. We also describe a variant of maxRPWC+ which is more efficient when applied during search due to the lighter use of data structures.

The second algorithm, called HOSTR⋆, extends the state-of-the-art GAC algorithm STR to a higher-order local consistency that can delete tuples from constraint relations as well as values from domains. HOSTR⋆ is actually a family of algorithms that combines the operation of STR (or a refinement of STR such as STR2) when establishing GAC on individual constraints and the operation of maxRPWC+ when trying to extend GAC supports to intersecting constraints. We describe several instantiations of HOSTR⋆ which differ in their implementation details.

We theoretically study the pruning power of the proposed algorithms and place them in a partial hierarchy which includes GAC, maxRPWC, and full pairwise consistency (FPWC), which is pairwise consistency followed by GAC. We show that the level of local consistency achieved by HOSTR⋆ is incomparable to that achieved by maxRPWC+ and to maxRPWC, but weaker than FPWC. Interestingly, a simple variant of HOSTR⋆ achieves FPWC, albeit with a high cost. maxRPWC+ achieves a consistency that is incomparable to maxRPWC but still stronger than GAC.

Experimental results from benchmark problems used in the evaluation of filtering algorithms for table constraints demonstrate that the proposed algorithms are considerably faster than the generic maxRPWC1 algorithm in classes of problems with large tables. Also, the best among the proposed algorithms outperforms the state-of-the-art GAC algorithm STR2 on some problem classes with intersecting table constraints, being orders of magnitude faster in many cases. However, it is also outperformed by STR2 on other problem classes with the differences being quite large in some cases.

We also compare our approaches to the state-of-the-art Abscon solver which is a solver with a more optimized implementation than our own. Results demonstrate that Abscon is faster on most problem classes, but since our methods often cut down the size of the search tree considerably, there are quite a few cases where this is reflected on important CPU time gains.

Finally, comparing our methods to eSTR2, a relevant later introduced algorithm [24], we observe that although eSTR2 is faster on most problems, its high memory requirements mean that its use is infeasible on certain instances. In contrast, our algorithms do not suffer from the drawback of high memory consumption and can easily handle the instances where eSTR2 fails.

The rest of this paper is structured as follows. Section 2 gives the necessary background. Section 3 presents algorithm maxRPWC+ and a variant of this algorithm that is more efficient when used during search. Section 4 extends STR-based algorithms to achieve stronger consistencies than GAC. In Section 5 we give experimental results. Finally, in Sections 6 and 7 we discuss related work and we conclude.

2 Background

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where:

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n variables.
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of finite domains, one for each variable, with maximum cardinality d.

- $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e constraints; with k denoting the maximum *arity* of the constraints.

Constraints can be defined either extensionally by listing the allowed (or disallowed) combinations of values or intensionally through a predicate or a function. A *positive table constraint* c_i is a constraint given in extension and defined by a set of allowed tuples. A tuple is called *allowed* iff it satisfies c_i. Each table constraint c_i is a pair $(scp(c_i), rel[c_i])$, where $scp(c_i) = \{x_{i_1}, \ldots, x_{i_r}\}$ is an ordered subset of $\mathcal{X}$ referred to as the *constraint scope*, and $rel[c_i]$ is a subset of the *Cartesian* product $D(x_{i_1}) \times \ldots \times D(x_{i_r})$ that specifies the allowed combinations of values (known also as c_i's relation) for the variables in $scp(c_i)$. For each table constraint $c_i \in \mathcal{C}$, we have $t_{c_i} = |rel[c_i]|$ and $t = maximum(t_{c_i})$.

An assigment of a value a_i to a variable x_i is denoted by (x_i, a_i). A tuple $\tau \in rel[c_i]$ is an ordered list of value to variable assignments $((x_1, a_1), \ldots, (x_m, a_m))$ s.t. $a_j \in D(x_j)$, $j = 1, \ldots, m$. Given a (table) constraint c_i, and a tuple $\tau \in rel[c_i]$, we denote by $\tau[x]$ the projection of τ on a variable $x \in scp(c_i)$ and by $\tau[X]$ the projection of τ on any subset $X \subseteq scp(c_i)$ of variables; $\tau[X]$ is called a *subtuple* of τ. A tuple is *valid* iff none of the values in the tuple has been removed from the variable domains of the corresponding variable. For any constraint c_i we denote by $\top$ (resp. $\bot$) a dummy tuple s.t. $\tau < \top$ (resp. $\tau > \bot$) for any tuple $\tau \in rel[c_i]$. We assume that for any table constraint its tuples are stored in lexicographical order.

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*. We denote by f_{min} the minimum number of variables that are common to any two constraints that intersect on more than one variable[1]. Therefore, f_{min} is at least two.

The most commonly used local consistency is *generalized arc consistency* (GAC) or *domain consistency*. A value $a_i \in D(x_i)$ is GAC iff for every constraint c, s.t. $x_i \in scp(c)$, there exists a valid tuple $\tau \in rel[c]$ s.t. $\tau[x_i] = a_i$ [30, 28]. In this case τ is a GAC-support of a_i on c. A variable is GAC iff all values in its domain are GAC. A problem is GAC iff there is no empty domain in $\mathcal{D}$ and all variables are GAC.

Strong Local Consistencies Several alternative local consistencies have been proposed, with domain filtering consistencies being especially interesting since they only remove values from domains and thus do not alter the structure of the constraint (hyper)graph or the constraints' relations. Examples of such consistencies for non-binary constraints include SAC [10], RPWC and maxRPWC [7], and rPIC [33]. Some of the consistencies, are directly defined on both binary and non-binary constraints (i.e., SAC for binary and non-binary constraints). Others, like maxRPWC, are defined on non-binary constraints but are inspired by relevant consistencies for binary constraints. A theoretical and experimental evaluation demonstrated that maxRPWC is a promising alternative to GAC [7].

A value $a \in D(x_i)$ is *max Restricted Pairwise Consistent* (maxRPWC) iff $\forall c_j \in C$, where $x_i \in scp(c_j)$, a has a GAC-support $\tau \in rel[c_j]$ s.t. $\forall c_l \in C$ $(c_l \neq c_j)$, s.t. $scp(c_j) \cap scp(c_l) \neq \emptyset, \exists \tau' \in rel[c_l]$, s.t. $\tau[scp(c_j) \cap scp(c_l)] = \tau'[scp(c_j) \cap scp(c_l)]$

[1] On constraints that intersect on one variable maxRPWC is equivalent to GAC [7].

and τ' is valid. In this case we say that τ' is a PW-support of τ and τ is a maxRPWC-support of a. A variable is maxRPWC iff all values in its domain are maxRPWC. A problem is maxRPWC iff there is no empty domain in $\mathcal{D}$ and all variables are maxRPWC.

GAC and maxRPWC are domain filtering (or first-order) consistencies, meaning that they can only identify inconsistent values, thereby filtering variable domains. Relation filtering (or higher-order) consistencies allow us to identify inconsistent tuples of values in constraints' relations. This is the case of *pairwise consistency* (PWC) defined as follows (for positive table constraints). A tuple τ_i in the table of a constraint c_i is PWC iff $\forall c_j \in \mathcal{C}$, $\exists \tau_j$ in $rel[c_j]$ which is a PW-support of τ_i. A problem is PWC iff every tuple of every constraint in the problem is PWC. We will also say that a problem is FPWC iff it is both PWC and GAC.

Following Debruyne and Bessiere [10], we call a local consistency A stronger than B iff in any problem in which A holds then B holds, and strictly stronger iff it is stronger and there is at least one problem in which B holds but A does not. Accordingly, A is incomparable to B iff none is stronger than the other.

Three algorithms for achieving maxRPWC have been proposed [7]. The first one, `maxRPWC1`, has O($e^2k^2d^p$) worst-case time complexity and O(ekd) space complexity, where p is the maximum number of variables involved in two constraints that share at least two variables. The second one has O(e^2kd^k) time complexity but its space complexity is exponential in p, and this can be prohibitive. The third one has the same time complexity as `maxRPWC1` but O(e^2kd) space complexity. Although `maxRPWC1` is less sophisticated than the other two, its performance when maintained during search is on average better than theirs because it uses lighter data structures. All these algorithms are generic in the sense that they do not consider any specific semantics that the constraints may have.

Recently, an extension of `STR`-based algorithms, called `eSTR`, that achieves full pairwise consistency was proposed [24]. This algorithm verifies the existence of a PW-support, in constant time, through the use of counters that store the number of occurrences of specific combinations of values in constraint intersections. As a result, the worst-case time complexity of one call to its basic filtering procedure (i.e., the revision of a constraint's table) is quite close to that of `STR` algorithms. Precisely, it is O($kd + max(k, g)t$), where g is the maximum number of non-trivial intersections that a constraint may have, while its worst-case space complexity of handling one constraint is O($n + max(k, g)t$). In practice, the space requirements, even for just one constraint, can be quite high because `eSTR` requires O(t) additional space for each intersecting constraint. A weaker version of `eSTR`, denoted by `eSTR`^{w}, was also proposed. This achieves a consistency level between GAC and FPWC and is more efficient in practice. Experiments demonstrated that this method can significantly outperform `STR2` in many problem classes, but it can become prohibitive when many and/or large intersections occur, due to its high space requirements.

3 Algorithm maxRPWC+

In this section we will first describe an algorithm that is based on maxRPWC and is specialized for application on table constraints, and then present an efficient variant of it that makes a lighter use of its data structures. The presented algorithm, called `maxRPWC+`, builds upon the generic maxRPWC algorithm `maxRPWC1` and a GAC algorithm that is specific to table constraints (called `GAC-va` hereafter) [23]. `maxRPWC+` not only specializes maxRPWC to table constraints but also introduces several techniques that help eliminate redundancies displayed by existing algorithms, such as unnecessary constraint checks and other operations on data structures. As in `GAC-va`, the main idea behind `maxRPWC+` is to interleave support and validity checks.

The approach of `GAC-va` involves visiting both lists of valid and allowed tuples in an alternating fashion when looking for a support (i.e., a tuple that is both allowed and valid). Its principle is to avoid considering irrelevant tuples by jumping over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. This is made possible because of the lexicographic ordering of tuples. The core operation of `GAC-va`, which is also exploited by our algorithm, is the construction of a valid tuple that is verified for being a GAC-support by searching for it in the list of allowed tuples using binary search. If it is not found, then the smallest allowed tuple that is greater than the aforementioned valid one is considered and its validity is checked. If it is not valid, then next valid tuple is constructed and so on.

Algorithm `maxRPWC+` uses the following data structures:

- For each constraint c and each value $a_i \in D(x_i)$, where $x_i \in scp(c)$, $allowed(c, x_i, a_i)$ is the list of allowed tuples in c that include the assignment (x_i, a_i).
- For each constraint c and each value $a_i \in D(x_i)$, where $x_i \in scp(c)$, $Last_{c,x_i,a_i}$ gives the most recently discovered (and thus lexicographically smallest) maxRPWC-support of a_i in c. The same data structure is used by `maxRPWC1` but it is exploited in a less sophisticated way as will be explained in Section 3.1.

Before going into the details of the algorithm we describe a simple modification that can be incorporated into any maxRPWC algorithm to boost its performance.

Restricted maxRPWC From the definition of maxRPWC we can see that the value deletions from some $D(x_i)$ may trigger the deletion of a value $b \in D(x_j)$ in two cases:

1. b may no longer be maxRPWC because its current maxRPWC-support in some constraint c is no longer valid and it was the last such support in c. We call this case *maxRPWC-support loss*.
2. The last maxRPWC-support of b in some constraint c may have lost its last PW-support in another constraint c' intersecting with c. We call this case *PW-support loss*.

Although detecting PW-support loss is necessary for an algorithm to achieve maxRPWC, our experiments have shown that the pruning it achieves rarely justifies its cost. Hence, `maxRPWC+` applies maxRPWC in a restricted way by only detecting

maxRPWC-support loss. However, the resulting method is still strictly stronger than GAC. This is clear if we consider that the "stronger" relationship is immediately derived by the definitions. Now consider a problem with constraints *alldiff*(x_1, x_2, x_3) and $x_1 = x_2$, and domains $\{0, 1, 2\}$ for all variables. This problem is GAC but the application of restricted maxRPWC will detect its inconsistency. Although a restricted version of maxRPWC is stronger than GAC, it obviously only achieves an approximation of maxRPWC. A similar approximation of the related binary local consistency maxRPC has also been shown to be efficient compared to full maxRPC [36, 1].

3.1 Algorithm description

Given a table constraint c_i, we now describe how algorithm `maxRPWC+` can be used to filter the domain of any variable $x_j \in scp(c_i)$. We assume that the domain of some variable in $scp(c_i)$ (different from x_j) has been modified and as a result the propagation engine will revise all other variables in $scp(c_i)$. Initially, Function 1 is called.

For each value $a_j \in D(x_j)$ Function 1 first searches for a GAC-support. This is done by calling function *seekSupport-va* which is an adaptation of Algorithm 12 of `GAC-va` [23]. This function makes an additional first check to verify if $Last_{c_i,x_j,a_j}$, which is the most recently found maxRPWC-support, and thus also GAC-support, is still valid. Note that the search starts from the first valid tuple. If $Last_{c_i,x_j,a_j}$ is valid, $\tau = Last_{c_i,x_j,a_j}$ is returned, else the valid and allowed tuples of c_i are visited in an alternating fashion. This is done by applying a dichotomic search in the list $allowed(c_i, x_j, a_j)$ to locate the lexicographically smallest valid and allowed tuple τ of c_i, such that $\tau > Last_{c_i,x_j,a_j}$ and $\tau[x_j] = a_j$. More precisely, τ can be either a valid tuple found in the list of allowed tuples of c_i or $\top$, in case of validity or support check failure. If such a tuple τ is found, we then check it for PW consistency through Function *isPWconsistent+* (Function 2). If a_j does not have a GAC-support (i.e., *seekSupport-va* returns $\top$) or none of its GAC-supports is a PW-support, then it will be removed from $D(x_j)$.

Function 1 *revisePW+* (c_i, x_j)

```
1: for each a_j ∈ D(x_j) do
2:    τ ←seekSupport-va(c_i, x_j, a_j);
3:    while τ ≠ ⊤ do
4:       if isPWconsistent+(c_i, τ) then break;
5:       τ ←seekSupport-va(c_i, x_j, a_j);
6:    if τ = ⊤ then remove a_j from D(x_j);
```

The process of checking if a tuple τ of a constraint c_i is PW consistent involves iterating over each constraint c_k that intersects with c_i on at least two variables and searching for a PW-support for τ (Function 2 line 1). For each such constraint c_k `maxRPWC+` first tries to quickly verify if a PW-support for τ exists by exploiting the $Last$ data structure as we now explain.

3.1.1 Fast check for PW-support

For each variable x_k belonging to the intersection of c_i and c_k, we check if $\tau' = Last_{c_k,x_k,\tau[x_k]}$ is valid and if it includes the same values for the rest of the variables in the intersection as τ (line 6 in Function 2). Function *isValid* simply checks if all values in the tuple are still in the domains of the corresponding variables. If these conditions hold for some variable x_k in the intersection then τ is PW-supported by τ'. Hence, we move on to the next constraint intersecting c_i.

Function 2 *isPWconsistent+* (c_i, τ): **boolean**

```
1:  for each c_k ≠ c_i s.t. |scp(c_k) ∩ scp(c_i)| > 1 do
2:     PW ← FALSE;
3:     max_τ' ← ⊥;
4:     for each x_k ∈ scp(c_k) ∩ scp(c_i) do
5:        τ' ← Last_{c_k,x_k,τ[x_k]};
6:        if isValid(c_k, τ') AND τ'[scp(c_k) ∩ scp(c_i)] = τ[scp(c_k) ∩ scp(c_i)] then
7:           PW ← TRUE; break;
8:        if τ' > max_τ' then max_τ' ← τ';
9:     if ¬PW then
10:       if seekPWSupport(c_i, τ, c_k, max_τ') = ⊤ then
11:          return FALSE;
12: return TRUE;
```

Else, we find max_τ' the lexicographically greatest $Last_{c_k,x_k,\tau[x_k]}$ among the variables that belong to the intersection of c_i and c_k and we search for a new PW-support in Function *seekPWSupport* (line 10). In case *seekPWSupport* returns $\top$ for some c_k then *isPWconsistent+* returns FALSE and a new GAC-support must be found and checked for PW consistency.

3.1.2 Fast check for lack of PW-support

Function 3 *seekPWsupport* $(c_i, \tau, c_k,$ max_τ')

```
1:  if ¬ isValid(c_k, max_τ') then max_τ' ← setNextTuple(c_i, τ, c_k, max_τ');
2:  if max_τ' ≠ ⊤ then τ' ← checkPWtuple(c_i, τ, c_k, max_τ');
3:  else return ⊤;
4:  x_ch ← select a variable ∈ scp(c_i) ∩ scp(c_k)
5:  while τ' ≠ ⊤ do
6:     τ'' ← binarySearch(allowed(c_k, x_ch, τ'[x_ch]),τ');
7:     if τ'' = τ' OR isValid(c_k, τ'') then return τ'';
8:     if τ'' = ⊤ then return ⊤;
9:     τ' ← setNextTuple(c_i, τ, c_k, τ'');
10: return ⊤;
```

Function 3 seeks a PW-support for τ in $rel[c_k]$. Before commencing with this search, it performs a fast check aiming at detecting a possible inconsistency (and thus

avoiding the search). In a few words, this check can sometimes establish that there cannot exist a PW-support for τ. This is accomplished by exploiting the lexicographical ordering of the tuples in the constraints' relations.

In detail, the validity of max_τ' is first checked in line 1. If *isValid* returns FALSE, then function *setNextTuple* is called to find the lexicographically smallest valid tuple in c_k that is greater than max_τ' and is such that $max\_\tau'[scp(c_k) \cap scp(c_i)] = \tau[scp(c_k) \cap scp(c_i)]$. If no such tuple exists, *setNextTuple* returns $\top$, and the search terminates since no PW-support for τ exists in c_k. If a tuple max_τ' is located then Function *checkPWtuple* is called to essentially perform a lexicographical comparison between max_τ' and τ taking into account the intersection of the two constraints (line 2 in Function 3). According to the result we may conclude that there can be no PW-support of τ in c_k and thus Function 3 will return $\top$. Consequently, lines 2-3 of Function 3 perform the fast check for lack of PW-support.

The addition of this simple check enables `maxRPWC+` to perform extra pruning compared to a typical maxRPWC algorithm. Before explaining how *checkPWtuple* works, we demonstrate this with an example.

Example 1 Consider a problem that includes two constraints c_1 and c_2 with $scp(c_1) = \{x_1, x_2, x_3, x_4\}$ and $scp(c_2) = \{x_3, x_4, x_5, x_6\}$. Assume that the GAC-support $\tau = (0, 2, 2, 1)$ has been located for value 0 of x_1 and that there exists a valid PW-support for τ in c_2 (e.g., $\{2, 1, 2, 2\}$). Also, assume that $Last_{c_2,x_3,2}$ and $Last_{c_2,x_4,1}$ are tuples $\tau' = (2, 2, 0, 1)$ and $\tau'' = (1, 1, 2, 3)$, meaning that $max\_\tau' = \tau'$. Since τ has a PW-support, a maxRPWC algorithm will discover this and will continue to check the next constraint intersecting c_1. However, since $\tau'[x_4]$ is greater than $\tau[x_4]$, it is clear that there is no PW consistent tuple in c_2 that includes values 2 and 1 for x_3 and x_4 respectively. If we assume that τ is the last GAC-support of $(x_1, 0)$ then `maxRPWC+` will detect this and will delete 0 from $D(x_1)$, while a maxRPWC algorithm will not.

Function 4 *checkPWtuple* $(c_i, \tau, c_k, max\_\tau')$

```
1: for each x_k ∈ scp(c_k) do
2:    if x_k ∉ scp(c_k) ∩ scp(c_i) then
3:       if max_τ'[x_k] is last value in D(x_k) then continue;
4:       else break;
5:    else
6:       if max_τ'[x_k] < τ[x_k] then break;
7:       if max_τ'[x_k] > τ[x_k] then return ⊤;
8: return max_τ';
```

Function *checkPWtuple* (Function 4) checks if there can exist a tuple greater or equal to max_τ' that has the same values for the variables of the intersection as τ. Crucially, this check is done in linear time as follows: Assuming $max\_\tau' = \langle (x_1, a_1), ..., (x_m, a_m) \rangle$ then this tuple is scanned from left to right. If the currently examined variable x_k belongs to $scp(c_k) \cap scp(c_i)$ and $a_k > \tau[x_k]$, where a_k is the value of x_k in max_τ', then we conclude that there can be no PW-support for τ in c_k (line 7). If x_k does not belong to $scp(c_k) \cap scp(c_i)$ then if the value it takes in max_τ' is the last value

in its domain, we continue searching (line 3). Otherwise, the scan is stopped because there may exist a tuple larger or equal to max_τ' that potentially is a PW-support of τ. However, max_τ' can still be used to avoid searching for a PW-support from scratch. Hence it is returned to *seekPWsupport*.

The soundness of the described process is guaranteed by the assumption that tuples in relations are stored in lexicographical order, which is typically the case. Given this assumption, it is certain that if a tuple τ which we try to extend to a PW-support is lexicographically smaller than max_τ', with respect to the values of the shared variables, then there can be no PW-support for τ. Otherwise, the lexicographical order would be violated.

3.1.3 Searching for PW-support

In case no inconsistency is detected through the fast check, then the search for a PW-support for τ begins, starting with the tuple τ' returned from *checkPWtuple*. We first check if τ' is an allowed tuple using binary search in a similar way to `GAC-va`. However, since there are more than one variables in the intersection of c_i and c_k, the question is which list of allowed tuples to consider when searching. Let us assume that the search will be performed on the list $allowed(c_k, x_{ch}, \tau'[x_{ch}])$ of variable x_{ch}. After describing the process, we will discuss possible criteria for choosing this variable.

Binary search will either return τ' if it is indeed allowed, or the lexicographically smallest allowed tuple τ'' that is greater than τ', or $\top$ if no such tuple exists. In the first case a PW-support for τ has been located and it is returned. In the third case, no PW-support exists. In the second case, we check if τ'' is valid, by using function *isValid* and if so, then it constitutes a PW-support for τ. Otherwise, function *setNextTuple* is called taking τ'' and returning the smallest valid tuple for $scp(c_k)$ that is lexicographically greater than τ'', such that $\tau'[scp(c_k) \cap scp(c_i)] = \tau[scp(c_k) \cap scp(c_i)]$ (line 9). If *setNextTuple* returns $\top$ the search terminates, otherwise, we continue to check if the returned tuple is allowed as explained above, and so on.

3.1.4 Selecting the list of allowed tuples

Since there are $|scp(c_k) \cap scp(c_i)|$ variables in the intersection of c_i and c_k, there is the same number of choices for the list of allowed tuples to be searched. Obviously, the size of the lists is a factor that needs to be taken into account. The selection of x_{ch} (line 4 of Function 3) can be based on any of the following (and possibly other) criteria:

1. Select the variable x_{ch} having minimum size of $allowed(c_k, x_{ch}, \tau'[x_{ch}])$.
2. Select the variable x_{ch} having the minimum number of tuples between τ' and $\top$ in $allowed(c_k, x_{ch}, \tau'[x_{ch}])$.
3. Select the leftmost variable in $scp(c_k) \cap scp(c_i)$.
4. Select the rightmost variable in $scp(c_k) \cap scp(c_i)$.

The first heuristic considers a static measure of the size of the lists. The second considers a more dynamic and accurate measure. In the experiments, presented below,

we have used the fourth selection criterion. Although this seems simplistic, as Example 2 demonstrates, there are potentially significant benefits in choosing the rightmost variable.

Example 2 Consider a constraint c_k on variables $x_1, \ldots, x_4$ with domains $D(x_1) = D(x_4) = \{0, \ldots, 9\}$ and $D(x_2) = D(x_3) = \{0, 1\}$. Assume that we are seeking a PW-support for tuple τ of constraint c_i in c_k. Also, $scp(c_k) \cap scp(c_i) = \{x_1, x_4\}$, $\tau[x_1] = 1$, $\tau[x_4] = 0$, and $|allowed(c_k, x_1, 1)| = |allowed(c_k, x_4, 0)|$. Figure 1 (partly) shows the lists $allowed(c_k, x_1, 1)$ and $allowed(c_k, x_4, 0)$. If we choose to search for a PW-support in $allowed(c_k, x_1, 1)$ then in the worst case binary search will traverse the whole list since tuples with value 0 for x_4 are scattered throughout the list. In contrast, if we choose $allowed(c_k, x_4, 0)$ then search can focus in the highlighted part of the list since tuples with value 1 for x_1 are grouped together.

x1	x2	x3	x4
1	0	0	0
....	...	...	...
1	0	0	9
1	1	0	0
....	...	...	...
1	1	0	9
....	...	...	...
1	1	1	9

x1	x2	x3	x4
0	0	0	0
....	...	...	...
0	1	1	0
1	0	0	0
....	...	...	...
1	1	1	0
....	...	...	...
9	1	1	0

Figure 1: $allowed(c_k, x_1, 1)$ and $allowed(c_k, x_4, 0)$.

3.2 Theoretical Results

We now analyze the worst-case complexity of the *revisePW+* function of `maxRPWC+`. The symbols M, N, S are explained in the proof.

Proposition 1 The worst-case time complexity of *revisePW+*(c_i, x_j) is O$(d.e.N.M(d+k.log(S)))$.

Proof: Let us first consider the complexities of the individual functions called by *seekPWsupport*. The cost of *setNextTuple* to construct a valid tuple for the variables that do not belong to the intersection is O$(d+(k-f_{min}))$. The cost of *checkPWtuple* is linear, since it requires at most O(k) checks to determine if any of $x_k \in c_k$ is inconsistent with $\tau[x_k]$. The worst-case time complexity of *binarySearch* is O$(k.log(S))$ with $S = |allowed(c_k, x_{ch}, \tau'[x_{ch}])|$. The worst-case time complexity for one execution of the loop body is then O$(d+(k-f_{min})+k.log(S))$=O$(d+k.log(S))$. Let us assume that M is the number of sequences of valid tuples that contain no allowed tuple, and for each tuple τ'' belonging to such a sequence $\tau''[scp(c_k) \cap scp(c_i)] = \tau[scp(c_k) \cap scp(c_i)]$. Then M bounds the number of iterations of the while loop in *seekPWsupport*. Therefore the worst time complexity of *seekPWsupport* is O$(M(d + k.log(S)))$.

The cost of *isPWconsistent+* is O$(e.M(d + k.log(S)))$, since in the worst case *seekPWsupport* is called once for each of the at most e intersecting constraints. The

maximum number of iterations for the while loop in *revisePW+* is N, where N is the number of sequences of valid tuples in c_i containing no allowed tuple. The cost of one call to *seekSupport-va* is O($d + k.log(S)$) [23]. Therefore, for d values the complexity of *revisePW+* is O($d.N(e.M(d + k.log(S)) + (d + k.log(S))))$=O($d.e.N.M(d + k.log(S))$).□

Note that M and N are at most $t_{c_k} + 1$ and $t_{c_i} + 1$ respectively, since in the worst case there is a sequence of valid tuples in between every pair of consecutive allowed tuples in a constraint's relation.

The complexity given by Proposition 1 concerns one call to *revisePW+* for one constraint. If *revisePW+* is embedded within an AC3-like algorithm (as `maxRPWC1` is) to achieve the propagation of all constraints in the problem then the worst-case time complexity of `maxRPWC+` will be O($e^2.k.d^2.N.M(d + k.log(S))$) since there are e constraints and each one is enqueued dk times in the worst case (i.e., once for each value deletion from a variable in its scope). Assuming the implementation of described by Lecoutre and Szymanek [23], the space complexity of `maxRPWC+` is O($e.k.|allowed(c, x, a)| + e.k.d$), where $|allowed(c, x, a)|$ is the maximum size of any constraint's relation and ekd is the space required for the $Last$ structure.

Regarding the pruning power of `maxRPWC+`, it is easy to show that the local consistency it achieves is strictly stronger than GAC (the arguments in the discussion on the pruning power of restricted maxRPWC are directly applicable). Also, the local consistency achieved by `maxRPWC+` is incomparable to maxRPWC. This is because a maxRPWC algorithm may achieve stronger pruning than `maxRPWC+` due to the detection of PW-support loss in addition to maxRPWC-support loss. On the other hand, the fast check for lack of PW-support enables `maxRPWC+` to prune extra values compared to maxRPWC.

Proposition 2 `maxRPWC+` achieves a local consistency that is incomparable to maxRPWC.

Proof: For an example where `maxRPWC+` achieves more pruning than a maxRPWC algorithm consider Example 1. For the opposite consider a problem with 0-1 domains that includes two constraints c_1 and c_2 with $scp(c_1) = \{x_1, x_2, x_3\}$ and $scp(c_2) = \{x_2, x_3, x_4\}$ having the allowed tuples $\{(0, 0, 0), (1, 0, 1), (1, 1, 0)\}$ and $\{(0, 0, 0), (0, 1, 1), (1, 0, 1)\}$ respectively. Now assume that value 0 is deleted from $D(x_4)$ which means that tuple $(0, 0, 0)$ of c_2 will be invalidated. `maxRPWC+` will revise all other variables involved in c_2 and only check for maxRPWC-support loss. Both values for x_2 and x_3 have maxRPWC-supports on c_2 so no deletion will be made. On the other hand, a maxRPWC algorithm will also check for PW-support loss by looking at constraint c_1. It will discover that value 0 of x_1 is no longer maxRPWC (its GAC-support has no PW-support on c_2) and will therefore delete it. □

3.3 A lighter version of maxRPWC+

Although `maxRPWC+` removes many redundancies that are inherent to generic algorithms through the exploitation of the $Last$ data structure, it suffers from an important drawback: the overhead required for the restoration of $Last$ after a failed instantiation. One way around this problem is to use $Last$ as a *residue*. That is, as a list of supports

that have been most recently discovered but are not maintained/restored during search. The resulting algorithm does not remove all the redundancies that maxRPWC+ does, but it is much cheaper to apply during search. This is similar to the relation between the optimal AC algorithm AC2001/3.1 [6] and the residue-based AC3^{rm} [21], as well as, the optimal maxRPC algorithm maxRPC3 and its corresponding residue-based version maxRPC3^{rm} [2].

The residue-based version of maxRPWC+, which hereafter is called maxRPWC+r, is an algorithm that exploits backtrack-stable data structures inspired from AC3^{rm} and maxRPC3^{rm} (rm stands for multidirectional residues). The $Last$ structure is not maintained incrementally as by maxRPWC+, but it is only used to store residues. As explained, a residue is a support which has been located and stored during the execution of the procedure that proves that a given tuple is maxRPWC. The algorithm stores the most recently discovered support, but does not guarantee that any lexicographically smaller value is not a maxRPWC-support. Consequently, when we search for a new maxRPWC-support in a table, we always start from scratch. $Last$ need not be restored after a failure; it can remain unchanged, hence a minimal overhead on the management of data.

To obtain the residue-based maxRPWC+r algorithm, we need to make the following simple modifications. In Function 2 we omit line 8, since $Last_{c_k,x_k,\tau[x_k]}$ may not be the lexicographically smallest tuple in c_k and thus, we cannot locate the max_τ' tuple. Subsequently, in Function 3 max_τ' is set to *setNextTuple*$(c_i, \tau, c_k, \bot)$ (namely the search for a PW-support in c_k starts from scratch). Additionally, the fast check for lack of PW-support, handled in *checkPWtuple* (line 2) is not feasible. Lines 1-3 of Function 3 are replaced with the following two:

1: $max\_\tau' \leftarrow$ *setNextTuple*$(c_i, \tau, c_k, \bot)$;

2: **if** $max\_\tau' = \top$ **then return** $\top$;

Regarding the time complexity of maxRPWC+r, the cost of calling Function *revisePW+* once is the same as in maxRPWC+. However, if we consider that repeated calls may be required due to the effects of constraint propagation then the complexity of maxRPWC+r for the whole problem is O$(e^2.k.d^3.N.M(d + k.log(S)))$.

Finally, regarding the pruning power of maxRPWC+r it is easy to see that this algorithm achieves a local consistency that is stronger than GAC (again the arguments in the discussion on the pruning power of restricted maxRPWC are directly applicable). Also, it achieves a local consistency weaker than maxRPWC+ since the two algorithms are essentially the same, minus the fast check for lack of PW-support.

4 Extending STR to a higher-order consistency

Algorithms based on STR, especially STR2 and STR3, have been shown to be the most efficient GAC algorithms for table constraints, along with the MDD approach of [9]. The idea of STR algorithms is to dynamically maintain the tables of supports while enforcing GAC, based on an optimization of simple tabular reduction (STR), a technique proposed by J. Ullmann [34].

4.1 The HOSTR* algorithm

In this section we present ways to extend STR algorithms in order to achieve more pruning than GAC. From now on we call these algorithms HOSTR$*$, which is derived from *higher-order* STR. The '$*$' stands for a particular STR algorithm (i.e., when extending STR2 we name the algorithm HOSTR2).

Algorithm 5 presents the main framework for HOSTR by extending the basic STR algorithm to achieve a stronger consistency. We choose to present an extension of STR as opposed to STR2 and STR3 because of STR's simplicity. STR2 and STR3 can be extended in a very similar way.

We now present the data structures used by STR and HOSTR.

- $rel[c]$: is the set of allowed tuples associated with a positive table constraint c. This set is represented by an array of tuples indexed from 1 to t_c which denotes the size of the table (i.e., the number of allowed tuples).
- $position[c]$: is an array of size t_c that provides indirect access to the tuples of c. At any given time the values in $position[c]$ are a permutation of $\{1, 2, \ldots, t_c\}$. The i^{th} tuple of c is $rel[c][position[c][i]]$. The use of this data structures enables restoration of deleted tuples in constant time.
- $currentLimit[c]$: is the position of the last current tuple in $rel[c]$. The current table of c is composed of exactly $currentLimit[c]$ tuples. The values in $position[c]$ at indices ranging from 1 to $currentLimit[c]$ are positions of the current tuples of c.
- $pwValues[x]$: is a set for each variable x, that contains all values in $D(x)$ which are proved to have a PW-support when HOSTR is applied on a constraint c. STR uses a similar data structure to store the values that are GAC-supported.
- X_{evt} is a set of variables. After HOSTR has finished processing a constraint c, X_{evt} will contain each variable $x \in scp(c)$ s.t. at least one value was removed from $D(x)$.

Propagation in STR based algorithms can be implemented by means of a queue that handles constraints. Once a constraint is removed from the queue, STR iterates over the valid tuples in the constraint until $currentLimit[c]$ is reached. STR removes any tuple that has become invalid through the deletion of one of its values (just like the **while** loop in Algorithm 5). Importantly, when a tuple is removed it still remains in $rel[c]$; its index is swapped in $position[c]$ with the index pointed by $currentLimit[c]$ (i.e., the index of the last valid tuple of $rel[c]$) in constant time. Additionally, the only information that is restored upon backtracking is $currentLimit[c]$ instead of the removed tuples. Thus, after finishing the **while** loop iteration, only valid and allowed tuples are kept in tables. Any value that are no longer supported, are deleted (the **for** loop in Algorithm 5).

HOSTR is identical to STR, except for the extra PW-check it applies when a tuple is verified as valid. To be precise, if an allowed tuple τ is proved valid by function *isValid* then HOSTR checks if it is also PW-consistent. This is done by calling the *isPWConsistent-STR* function (line 7).

We omit the detailed description of functions *isValid* and *removeTuple*, which can be found in [18]. Briefly, *isValid* takes a tuple τ and returns true iff none of the values in τ has been removed from the domain of the corresponding variable. *removeTuple* again takes a tuple τ and removes it in constant time by replacing $position[c][i]$, where i is the position of τ in $rel[c]$, with $position[c][currentLimit[c]]$ (namely by swapping indexes and not tuples) and then decrementing $currentLimit[c]$ by one. As a result this procedure violates the lexicographic ordering of the remaining tuples in $position[c]$.

Algorithm 5 *HOSTR*$\star$ (c: constraint): set of variables

```
 1: for each unassigned variable x ∈ scp(c) do
 2:    pwValues[x] ← ∅;
 3: i ← 1;
 4: while i ≤ currentLimit[c] do
 5:    index ← position[c][i];
 6:    τ ← rel[c][index];
 7:    if isValid(c, τ) AND isPWconsistent-STR(c,τ) then
 8:       for each unassigned variable x ∈ scp(c) do
 9:          if τ[x] ∉ pwValues[x] then
10:             pwValues[x] ← pwValues[x] ∪{τ[x]};
11:       i ← i + 1;
12:    else
13:       removeTuple(c, i); // currentLimit[c] decremented
   // domains are now updated and X_evt computed
14: X_evt ← ∅;
15: for each unassigned variable x ∈ scp(c) do
16:    if pwValues[x] ⊂ D(x) then
17:       D(x) ← pwValues[x];
18:       if D(x) = ∅ then
19:          throw INCONSISTENCY;
20:       X_evt ← X_evt ∪ {x};
21: return X_evt;
```

Once a tuple τ of constraint c has been verified as valid, Function *isPWconsistent-STR* of HOSTR is called. This function iterates over each constraint c_k that intersects with c on at least two variables and searches for a PW-support for τ. If τ has no PW-support on some c_k then it will be removed in line 13 of Algorithm 5.

Once the traversal of the valid tuples has terminated, HOSTR (and STR) updates the X_{evt} set to include any variable that belongs to $scp(c)$ and has had its domain reduced (lines 14-21). Thereafter, all constraints that involve at least one variable in X_{evt} will be added to the propagation queue.

Concerning the implementation of *isPWconsistent-STR*, there are several options, with each one resulting in a different variant of HOSTR. The following variants differ in the way search for a PW-support is implemented.

1. **Linear**: For each constraint c_k *isPWconsistent-STR*^{l} iterates in a linear fashion over each τ' that currently belongs to $rel[c_k]$ to locate a valid tuple such that

$\tau'[scp(c_k) \cap scp(c)] = \tau[scp(c_k) \cap scp(c)]$, until $currentLimit[c_k]$ is reached. This version does not require the use of any additional data structures.

2. **Binary**: For each constraint c_k *isPWconsistent-STR*^{b} locates a τ' by applying a binary search on the initial $rel[c_k]$. Then it checks its validity and whether it satisfies the condition $\tau'[scp(c_k) \cap scp(c)] = \tau[scp(c_k) \cap scp(c)]$. A requirement of this version is that the original table is stored. Note that binary search on the current tables as they are stored by STR is not possible since the lexicographic ordering of the tuples is violated.

3. **STR/maxRPWC+ Hybrid**: In addition to the data structures of STR, this version keeps the $allowed(c, x, a)$ lists of maxRPWC+ (see Section 3). Then, following maxRPWC+, for each constraint c_k *isPWconsistent-STR*^{h} visits the lists of valid and allowed tuples in an alternating fashion, using binary search, to locate a valid tuple τ' such that $\tau'[scp(c_k) \cap scp(c)] = \tau[scp(c_k) \cap scp(c)]$.

Hereafter, we denote by HOSTR^l (respectively HOSTR^b, HOSTR^h) the HOSTR algorithm that utilizes the function *isPWconsistent-STR*^{l} (respectively *isPWconsistent-STR*^{b}, *isPWconsistent-STR*^{h}).

In practice, the STR/maxRPWC+ hybrid approach (i.e., HOSTR^h) is by far the most efficient among the above methods. *isPWconsistent-STR*^{h} which implements this method closely follows the operation of maxRPWC+ when looking for a PW-support, minus the usage of the $Last$ structure. This is displayed in Function 6. For each c_k *isPWconsistent-STR*^{h} calls function *setNextTuple* to find the lexicographically smallest valid tuple in c_k, such that $\tau'[scp(c_k) \cap scp(c)] = \tau[scp(c_k) \cap scp(c)]$. The search in c_k (line 2) starts from scratch (i.e., $\bot$), since HOSTR does not store information about recently found maxRPWC-supports. If no such tuple exists, *setNextTuple* returns $\top$, and the search terminates since no PW-support for τ exists in c_k. Else, we check if τ' is an allowed tuple using binary search as explained in Function 3.

Function 6 *isPWconsistent-STR*$^h(c, \tau)$: **boolean**

```
1: for each c_k ≠ c s.t. |scp(c_k) ∩ scp(c)| > 1 do
2:     τ' ← setNextTuple(c, τ, c_k, ⊥);
3:     x_ch ← select a variable ∈ scp(c) ∩ scp(c_k);
4:     while τ' ≠ ⊤ do
5:         τ'' ← binarySearch(allowed(c_k, x_ch, τ'[x_ch]),τ');
6:         if τ'' = τ' OR isValid(c_k, τ'') then return TRUE;
7:         if τ'' = ⊤ then return FALSE;
8:         τ' ← setNextTuple(c, τ, c_k, τ'');
9: return TRUE;
```

4.2 Theoretical Results

For a given constraint c, the worst-case time complexity of STR is O($k'd + kt'$), where k' denotes the number of uninstantiated variables in $scp(c)$ and t' denotes the size

of the current table of c [18]. The worst-case space complexity of STR is O(ekt). We now give the worst-case time complexity of HOSTR when implemented using the STR/maxRPWC+ Hybrid approach.

Proposition 3 The worst-case time complexity of HOSTR for the processing of one constraint is O($k'.d + k.t'(e.M(d + k.log(S)))$).

Proof: HOSTR is identical to STR with the addition of the call to *isPWconsistent-STR*^{h} each time a valid tuple is verified. The time complexity of Function *isPWconsistent-STR*^{h} is O($e.M(d + k.log(S))$) (see the proof of Proposition 1 for details). Therefore, the worst-case time complexity of HOSTR is O($k'.d + k.t'(e.M(d + k.log(S)))$). □

If HOSTR is embedded within an AC-3 like algorithm to propagate all constraints then the time complexity will be O($e.d.k.(k'.d + k.t'(e.M(d + k.log(S))))$) since there are e constraints and each one is enqueued dk times in the worst case (i.e., once for each value deletion from a variable in its scope). The space complexity of HOSTR is O($e(kt + k.|allowed(c, x, a)|)$) since in addition to the original tables we need the $allowed(c, x, a)$ lists.

Now we prove that HOSTR, implemented in any of the above ways, achieves a local consistency that is incomparable to that achieved by maxRPWC+ and to maxRPWC, and is weaker than FPWC.

Proposition 4 The local consistency achieved by HOSTR is incomparable to that achieved by maxRPWC+.

Proof: First, consider the problem in Example 1. Since max_τ' is lexicographically greater than the PW-support of τ on c_2 (let us assume that this is the only PW-support for τ), this tuple must be PW inconsistent. This means that when HOSTR processes c_2 it will delete this tuple. If, however, this does not cause any value deletions then it will not be propagated any further and c_1 will not be processed. Hence, HOSTR will not be able to delete 0 from $D(x_1)$. As explained in Example 1, maxRPWC+ will be able to make this deletion.

To show that HOSTR can delete values that maxRPWC+ cannot delete, consider the problem depicted in Figure 2. There are five variables $x_1, \ldots, x_5$ with $\{0, 1\}$ domains and one variable (x_6) with domain 0. There are three table constraints with their allowed tuples shown in Figure 2. Value 0 of x_1 has tuple (0,0,0) as GAC-support in $rel[c_1]$. This tuple has the PW-support (0,0,0,0) in $rel[c_2]$, and therefore if maxRPWC+ is applied it will not delete it (as it will not delete any other value). Now assume that HOSTR processes c_2 first. Tuple (0,0,0,0) does not have a PW-support in c_3, and therefore it will be removed. When c_1 is processed, HOSTR will determine that tuple (0,0,0) has no PW-support in c_2, since (0,0,0,0) has been deleted, and will therefore be removed. As a result, value 0 of x_1 will loose its only GAC-support in c_1 and will be deleted. □

The extra pruning achieved by HOSTR compared to maxRPWC+ in the above example is a direct consequence of the fact that HOSTR, like STR, removes tuples from constraint relations. Now note that if constraint c_1 is processed first in the example then no value deletion will be made. This is because when c_1 is processed tuple (0,0,0) in $rel[c_1]$ will have the PW-support (0,0,0,0) in $rel[c_2]$ and therefore value 0 of x_1 will not be deleted. When c_2 is later processed, tuple (0,0,0,0) will indeed be removed but

no value from any of the variables in $scp(c_2)$ will be deleted. This means that X_{evt} will be empty and as a result no constraint will be added to the propagation queue. Therefore, c_1 will not be processed again. Hence, the pruning power of HOSTR cannot be characterized precisely because it depends on the ordering of the propagation queue.

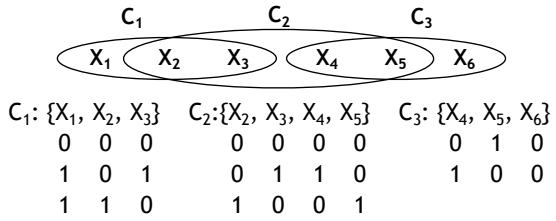

Figure 2: HOSTR vs. maxRPWC+.

Proposition 5 The local consistency achieved by HOSTR is incomparable to maxRPWC.

Proof: First, consider the example of Figure 2. If maxRPWC is applied on this problem it will achieve no pruning. But as explained in the proof of Proposition 4, HOSTR will delete value 0 of x_1 if c_2 is processed before c_1. Therefore, HOSTR can achieve stronger pruning than a maxRPWC algorithm.

Now consider the example in the proof of Proposition 2. After the deletion of value 0 from $D(x_4)$, HOSTR will add x_4 to X_{evt} and enqueue c_2. When c_2 is then processed no value deletion will be made and therefore propagation will stop. On the other hand, as explained, a maxRPWC algorithm will delete value 0 from $D(x_1)$. □

Proposition 6 HOSTR achieves a local consistency that is strictly weaker than PWC+GAC.

Proof: We first show that any deletion made by HOSTR will also be made by an algorithm that applies FPWC. HOSTR will delete a tuple $\tau \in rel[c]$ if τ is invalid or if Function *isPWconsistent-STR*^{h} cannot find a PW-support for τ on some constraint c'. By definition, applying PWC deletes any value that is invalid or not PWC. Hence, it will also delete τ. Correspondingly, HOSTR will delete a value $a \in D(x)$ if it does not participate in any valid and PW consistent tuple on some constraint c that involves x. Considering an algorithm that applies FPWC, the PWC phase will delete all invalid and PW inconsistent tuples from $rel[c]$. Hence, the application of GAC that follows will delete a.

Finally, consider the example in Figure 2. As explained, if c_1 is processed before c_2, HOSTR will achieve no pruning. In contrast, and independent of the order in which constraints are processed, the application of PWC will remove tuple (0,0,0,0) from $rel[c_2]$ and because of this tuple (0,0,0) will be removed from $rel[c_1]$. Then when GAC is applied value 0 will be removed from $D(x_1)$ because it will have no support in c_1. Hence, FPWC is strictly stronger than HOSTR. □

4.3 A stronger version of HOSTR

Motivated by the inability to precisely characterize the pruning power of HOSTR due to its dependence on the propagation order, we propose a simple modification to HOSTR which achieves a stronger consistency property that can be precisely characterized. This algorithm, which we call full HOSTR (fHOSTR) differs from HOSTR in the following: If after traversing the tuples of a constraint c, at least one tuple has been removed then all variables that belong to $scp(c)$ are added to X_{evt}. This means that all constraints that involve any of these variables will be then added to the propagation queue. Recall that HOSTR adds a variable to X_{evt} only if a value has been deleted from the domain of this variable.

It is easy to see that in the example in Figure 2 fHOSTR will make the same value and tuple deletions as a FPWC algorithm. Generalizing this, we now prove that fHOSTR achieves the level of consistency of FPWC.

Proposition 7 Algorithm fHOSTR achieves FPWC.

Proof: We show that any deletion made by applying FPWC will also be made by fHOSTR. Consider any value a that is removed from a domain after FPWC is applied. This is because all supports for this value on some constraint c have been deleted. These tuples were deleted because they are not valid or not PWC. In the former case, since fHOSTR fully includes the operation of STR, it will delete any invalid tuple when processing constraint c and therefore will also delete value a. In the latter case, consider the deletion of any tuple τ because it is not PWC. This means that all of τ's PW-supports on some constraint c' have been deleted. When processing c', once the last PW-support of τ is deleted, fHOSTR will enqueues all constraints that intersect with c', including c. Then when c is processed, fHOSTR will not be able to find a PW-support for τ on c' and will thus delete it. Hence, all support of a will be deleted and as a result a will be deleted. □

To complete the theoretical analysis of the algorithm's pruning power, we now show that fHOSTR achieves a local consistency that is strictly stronger than that achieved by maxRPWC+.

Proposition 8 The local consistency achieved by fHOSTR is strictly stronger than that achieved by maxRPWC+.

Proof: First we show that any deletion made by maxRPWC+ will be also made by fHOSTR. Consider a value a that is removed from $D(x)$ after maxRPWC+ is applied. This is because either:

1. a is not GAC,
2. no GAC-support of a on a constraint c has a PW-support on some constraint c_k,
3. the deletion is triggered by the "fast check for lack of PW-support".

In the first case, the STR step of fHOSTR will obviously discover that a is not GAC. In the second case, the lack of PW-support for a tuple τ that includes a on some constraint c_k means that no potential PW-support for τ is valid. When fHOSTR processes tuple τ

of c it will try to extend it to a PW-support in all intersecting constraints. Hence, it will consider c_k and recognize that τ has no valid PW-support in c_k. The same argument holds for all the GAC-supports of a in c.

Finally, if a is deleted by the "fast check for lack of PW-support" then the following must hold for each PW-consistent GAC-support τ for a in some constraint c. Any PW-support τ' of τ in some c_k is lexicographically smaller than $Last_{c_k,x_i,b}$, where $x_i \in scp(c_k)$ is a variable that belongs to the intersection of c and c_k and b is its value in τ'. maxRPWC+ must have moved the pointer $Last_{c_k,x_i,b}$ to some tuple τ'' beyond the PW-supports of τ because when trying to find a PW consistent GAC-support for b in c_k, all tuples lexicographically smaller than τ'', including all the PW-supports of τ, were determined as PW inconsistent. Now when fHOSTR processes c_k it will determine that all the PW-supports of τ are not themselves PW consistent and will thus delete them. Therefore, all constraints that involve variables in $scp(c_k)$, including c, will be enqueued. When c is later processed, no PW-support for τ (or any of a's GAC-supports in general) will be found, and thus a will be deleted.

For an example where fHOSTR achieves stronger pruning than maxRPWC+ consider the second example in the proof of Proposition 4. □

Algorithm fHOSTR achieves a stronger local consistency than maxRPWC+ and HOSTR but has a serious drawback: Its time complexity, when embedded within an AC-3 like algorithm to propagate all constraints, is $O(e^2.t.(k'.d+k.t'(e.M(d+k.log(S)))))$ since there are e constraints and each constraint c is enqueued $O(et)$ times in the worst case (i.e., once for each tuple deletion from a constraint intersecting c). This complexity is prohibitive for large table constraints.

Figure 3 summarizes the relationships between the algorithms discussed throughout this paper with respect to their pruning power. Note that by GAC, maxRPWC, and FPWC we mean any algorithm that achieves these local consistency properties.

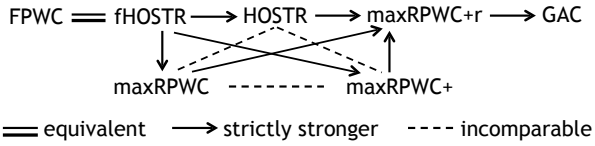

Figure 3: Summary of the relationships between algorithms.

5 Experiments

We ran experiments on benchmark non-binary problems with table constraints from the CSP Solver Competition[2]. The arities of the constraints in these problems range from 3 to 18. We tried the following classes: *forced random problems*, *random problems*, *positive table constraints*, *BDD*, *Dubois*, and *Aim*. These classes represent a large spectrum of instances with positive table constraints that are very commonly used for

[2]http://www.cril.univ-artois.fr/CPAI08/

the evaluation of GAC algorithms and additionally, non-trivial intersections exist between their constraints. The first two classes only include constraints of arity 3, while the others include constraints of large arity (up to 18). Note that there exist classes of problems with table constraints where maxRPWC and similar methods do not offer any advantage compared to GAC because of the structure of the constraints. For example, on crossword puzzles constraints intersect on at most one variable. Our algorithms cannot achieve extra filtering compared to GAC in such problems and thus we have not included them in this study.

In more detail, the 150 tried instances belong to classes that have the following attributes:

- The first two series *Random-fcd* and *Random* involve 20 variables and 60 ternary relations of almost 3,000 tuples each.
- The series *BDD* involves 21 Boolean variables and 133 constraints generated from binary decision diagrams of arity 18 that include 58,000 tuples.
- The two series *Positive table-8* and *Positive table-10* contain instances that involve 20 variables. Each instance of the series *Positive table-8* (resp.,*Positive table-10*) involves domains containing 5 (resp. 10) values and 18 (resp. 5) constraints of arity 8 (resp. 10). The constraint tables contain about 78,000 and 10,000 tuples, respectively.
- The *Dubois* class contains instances involving 80 Boolean variables and quaternary constraints.
- The *Aim-100* and *Aim-200* series involve 100 and 200 Boolean variables respectively, with constraints of small arities (mainly ternary and a few binary).

The algorithms 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 for all algorithms and all instances. Search used the *dom/ddeg* heuristic for variable ordering and lexicographical value ordering. We have chosen *dom/ddeg* [5] as opposed to the generally more efficient *dom/wdeg* [8] 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 [3]. Having said this, experiments with *dom/wdeg* did not give significantly different results compared to *dom/ddeg* as far as the relative efficiency of the algorithms is concerned.

We present results from our baseline implementations of `STR2` and `maxRPWC1`, upon which the new algorithms were built, compared to the proposed algorithms, $\texttt{HOSTR2}^h$, `maxRPWC+` and `maxRPWC+r`. For `maxRPWC1` we used its residual version in order to avoid the maintenance of the $LastGAC$ structure during search. This resulted in faster run times. We also include results from our later designed algorithm $\texttt{eSTR2}^w$. We do not include results from $\texttt{fHOSTR2}^h$, since its restricted version, that approximates FPWC, is always more efficient. The differences between $\texttt{fHOSTR2}^h$ and $\texttt{HOSTR2}^h$ in favour of the latter range from being marginal to 3 times faster. As explained in Section 4.3 this is due to the high cost of applying the full algorithm.

Finally, we also compare our algorithms to the state-of-the-art solver Abscon[3]. Specifically, we compare against Abscon's implementations of the GAC algorithms `GAC-va` and `STR2`, denoted by `GAC-va_abs` and `STR2_abs` respectively. Although Abscon's implementation is much more optimized than ours, these results help to put our contributions in context of modern CP solvers and point out classes of problems where our methods are very effective.

5.1 Preprocessing

Table 1 shows the mean CPU times (in milliseconds) obtained by the tested algorithms on each problem class for the initialization (i) and the preprocessing (p) phase. During initialization, the data structures of a specific algorithm are initialized, while preprocessing includes one run of a specific filtering algorithm before the search commences. Therefore, initialization time is the time required for each algorithm to construct all its structures, while preprocessing time is the time for a stand alone use of a specific algorithm.

Our implementation for `STR2` is the fastest algorithm in the initialization phase in all tested classes, while both Abscon's GAC algorithms are faster in the preprocessing phase. It is notable that `STR2_abs` and `GAC-va_abs` require at least one order of magnitude more time to construct their structures compared to our `STR2`, hinting that they utilize intricate data structures that are later exploited throughout search (e.g., for fast restoration upon backtracking).

Table 1: Mean CPU times of the initialization (i) and the preprocessing (p) phase in milliseconds from various problem classes.

Problem Class		STR2	STR2_abs	GAC-va_abs	maxRPWC1	HOSTR2^h	maxRPWC+r	maxRPWC+	eSTR2^w
Random-fcd	i	**31**	646	678	115	345	202	263	689
	p	111	15	**0**	696	399	154	250	97
Random	i	19	644	677	**10**	249	249	272	616
	p	87	17	**0**	738	291	186	247	85
Positive table-8	i	83	2,654	3,280	**5**	1,305	1,510	1,628	76,062
	p	271	82	**0**	2,891	36,621	343	359	952
Positive table-10	i	2	601	653	**0**	236	263	304	12,214
	p	47	34	**2**	4,997,817	363,000	620,210	772,193	20
BDD	i	237	697	755	**123**	10,017	8,530	8,334	mem
	p	1,415	622	**6**	477,497	2,218	6,159	16,875	mem
Dubois	i	**10**	423	415	12	13	**10**	12	**10**
	p	**0**	2	**0**	5	**0**	2	2	2
Aim-100	i	**108**	431	435	126	244	111	195	213
	p	**2**	3	**2**	160	19	19	38	12
Aim-200	i	397	513	492	303	465	**270**	280	319
	p	4	**1**	4	174	30	53	97	11

Algorithms for strong local consistencies, as expected, spend extra time to record the intersections of the constraints during the initialization, while they are more expensive when they are applied stand-alone (i.e., during preprocessing). Interestingly, `maxRPWC1` has low initialization times, since it does not use the $allowed$ data structure that most proposed strong local consistency algorithms use. On the contrary,

[3]http://www.cril.univ-artois.fr/∼lecoutre/software

maxRPWC1 is by far inferior compared to all other algorithms during preprocessing. Particularly, on *Positive table-8* and *Positive table-10* it is more than two orders of magnitude worse than all proposed algorithms.

Regarding the initialization times of the three proposed strong consistency algorithms, maxRPWC+r appears to be the fastest. HOSTR2^h needs more time for the initialization since it uses the structures of both STR2 and maxRPWC+. Concerning the preprocessing, results are more varied since there are classes where maxRPWC+r dominates HOSTR2^h and vice versa. This is due to the different approaches of STR-like and GAC-va-like algorithms, as the former iterate over tuples and the latter over values and tuples, thus it is not clear which one is preferable for stand-alone use. On the other hand, it is clear that maxRPWC+r, being lighter, is faster than maxRPWC+.

eSTR2^w is the fastest algorithm during preprocessing among the strong algorithms. Particularly, its times are very close to the ones of STR2 due to their close time-complexity. On the contrary, it requires more time for the initialization phase in order to build its structures. This is evident especially on *Positive table-8* and *Positive table-10* because the tables are long and constraints intersect on many variables (7 and 8 respectively). On *BDD*, where constraints intersect on 16 variables, eSTR2^w reveals its main drawback as it exhausts all of the available memory.

Finally, the high preprocessing times for all proposed methods on the *Positive table-10* class are due to the high memory consumption on these large instances. However, as we show below, preprocessing by these algorithms is able to determine the unsatisfiability of the instances without requiring search. However, on *Positive table-8* both maxRPWC+ and maxRPWC+r are very close to STR2 and two (resp. one) orders of magnitude faster compared to HOSTR2^h (resp. maxRPWC1) preprocessing times.

5.2 Search

In Table 2 we present selected representative results from search algorithms that apply the tested filtering algorithms throughout search, while in Table 3 we give the mean and median performance of the search algorithms in each problem class[4]. Note that results from class *Aim-200* were obtained using the *dom/wdeg* variable ordering heuristic. This is because our algorithms were unable to solve these problems within the time limit using *dom/ddeg*.

All of the tried algorithms completed all instances within the cutoff limit except for maxRPWC1, which did not solve 13 instances out of 20 from the *Positive table-8* class. Also, eSTR2^w solved none of the *BDD* instances since it exhausted the available memory on each one of them.

Among the three proposed algorithms, maxRPWC+r is the most efficient with HOSTR2^h being a close second. maxRPWC+r is faster than HOSTR2^h on *Random*, *Random-fcd* and especially on *Positive table-8* (e.g., on the rand-8-20-5-18-800-12 instance it is over 6 times faster), while it is slower on *Positive table-10*. On the rest of the classes HOSTR2^h is better than maxRPWC+r, but without considerable differences. maxRPWC+r is also constantly faster compared to maxRPWC+, with the differences being considerable in the *Dubois* and *Aim* classes.

[4]These results include initialization and preprocessing times.

Comparing our algorithms against maxRPWC1 we observe that they are particularly efficient on instances of large arities. These are the instances of the *Positive Table* and *BDD* classes, whose arities vary from 8 to 18. Both maxRPWC+r and maxRPWC+ are notably faster, especially on *Positive table-10* and *BDD* they are superior by over one order of magnitude. There are also cases (i.e., *Positive table-8*), where maxRPWC1 was not able to solve the majority of the instances within the cutoff limit.

Table 2: CPU times (t) in secs and nodes (n) from various representative problem instances. A slash (-) means that the instance was not completed within the time limit. mem means that memory was exhausted.

Instance		STR2	STR2_abs	GAC-va_abs	maxRPWC1	HOSTR2^h	maxRPWC+r	maxRPWC+	eSTR2^w
rand-3-20-20	t	242	53	**30**	308	291	237	195	79
60-632-fcd-5	n	160,852	156,394	156,394	**66,335**	66,601	66,469	66,585	86,110
rand-3-20-20	t	90	20	**13**	174	131	78	82	31
60-632-fcd-7	n	73,536	72,178	72,178	19,680	19,791	19,988	18,668	**16,432**
rand-3-20-20	t	472	178	**91**	867	879	567	564	200
60-632-5	n	501,583	479,945	479,945	152,712	152,763	153,138	152,892	**150,893**
rand-3-20-20	t	16	6	**4**	32	38	18	17	11
60-632-14	n	19,996	16,511	16,511	3,976	3,986	4,002	3,898	**3,578**
rand-8-20-5	t	17	8	**7**	1,616	1,860	494	754	246
18-800-7	n	17,257	19,341	19,341	3,424	3,444	3,447	3,430	**1,001**
rand-8-20-5	t	19	**12**	16	-	12,625	2,823	3,654	147
18-800-12	n	105,521	95,895	95,895	-	28,830	28,752	28,662	**15,517**
rand-10-20	t	**0.4**	0.6	6	3,811	174	203	208	11
10-5-10000-1	n	1,110	1,111	1,111	**0**	**0**	**0**	**0**	**0**
rand-10-20	t	**0.3**	0.6	2.5	6,438	1,283	1,212	1,298	10
10-5-10000-4	n	1,110	1,110	1,110	**0**	**0**	**0**	**0**	**0**
bdd-21-133	t	30	11	18	2.4	**0.6**	1.5	2	mem
18-78-6	n	20,582	20,617	20,617	**0**	**0**	**0**	**0**	-
bdd-21-133	t	39	11	21	1,714	**1.2**	11.6	16.8	mem
18-78-11	n	19,364	19,670	19,670	**21**	**21**	**21**	**21**	-
dubois-21	t	110	69	49	56	**40**	53	314	50
	n	58M	56M	56M	**19M**	23M	23M	23M	23M
dubois-26	t	4,044	2,898	1,799	3,427	**1,463**	1,830	12,174	2,183
	n	1,823M	2,147M	2,147M	808M	**744M**	**744M**	**744M**	**744M**
aim-100	t	6,423	336	169	0.4	**0.15**	0.18	0.25	0.2
1-6-sat-2	n	29M	59M	59M	**100**	**100**	**100**	**100**	**100**
aim-100	t	14,862	185	**99**	3,378	5,211	3,766	15,617	3,334
2-0-sat-4	n	1,729M	**20M**	**20M**	51M	149M	149M	149M	149M
aim-200	t	57	9	3.6	0.5	**0.4**	0.5	0.6	0.9
2-0-sat-1	n	2M	575,943	575,943	**257**	2,210	2,210	1,782	9,847
aim-200	t	**0.7**	2.2	1.3	2.7	2.5	5.9	1.3	1.8
2-0-unsat-1	n	23,715	97,558	97,558	**16,091**	46,893	34,671	34,671	41,177

eSTR2^w is superior to the proposed algorithms by a factor of 3 on *Random* and *Random-fcd* while on *Aim* it displays a close performance. Also, it is one order of magnitude faster on *Positive table-8* and *Positive table-10*. These results are not surprising given than eSTR2^w is a more recent algorithm based on a different concept than the algorithms presented here. However, as noted before, eSTR2^w suffers from a serious drawback: its high memory consumption in the presence of large constraint intersections. Hence, on the *BDD* class it fails to even construct its data structures. This is because in this class constraints intersect on 16 variables and along with the big size of the tables the use of eSTR2 becomes prohibitive. In such kind of problems the need for algorithms with lighter data structures is mandatory and demonstrates the practical advantages of our approaches.

Now comparing Abscon to our algorithms, both STR2_abs and GAC-va_abs are much faster on the *Random* and *Positive table* classes. Counterwise, our algorithms typically outperform Abscon on *BDD* and *Dubois*, even by orders of magnitude in some cases (e.g., HOSTR2^h on bdd-21-133-18-78-6). This is the class where none of the GAC algorithms could detect unsatisfiability through preprocessing unlike strong consistency algorithms. In comparison with our implementation of STR2, the differences are even larger in favor of HOSTR2^h and maxRPWC+r on the same classes, albeit they are significantly outperformed on *Positive table* instances. We believe that our approaches will become more efficient and competitive as soon as they are integrated in an optimized implementation, such as that of Abscon. Albeit, even with the current implementation, they display important gains in many cases (i.e., in *BDD*, *Dubois*, and *Aim* instances).

Table 3: Mean CPU times (t) in secs, mean node visits (n), median CPU times (t_m) and median node visits (n_m) from various problem classes.

Class		STR2	STR2_abs	GAC-va_abs	maxRPWC1	HOSTR2^h	maxRPWC+r	maxRPWC+	eSTR2^w
Random	t	152	43	**28**	310	305	176	171	71
-fcd	n	132,492	156,326	156,326	49,009	49,317	49,420	48,608	**45,272**
	t_m	131	28	**18**	214	256	126	117	51
	n_m	105,423	95,995	95,995	31,399	31,503	31,610	30,727	**27,222**
Random	t	54	63	**39**	461	501	298	288	116
	n	211,456	215,763	215,763	79,928	80,172	80,316	78,802	**73,533**
	t_m	89	36	**18**	269	318	157	155	75
	n_m	102,327	95,417	95,417	41,243	41,301	41,322	41,107	**40,715**
Positive	t	15	**9**	11	-	5,002	1,168	1,576	134
table-8	n	52,313	46,478	46,478	-	10,087	9,787	10,039	**4,818**
	t_m	18	**10**	11	2,416	4,194	1,089	1,470	133
	n_m	45,880	39,923	39,923	7,721	8,134	7,979	8,112	**3,274**
Positive	t	**0.3**	0.8	3.5	4,998	363	620	773	12
table-10	n	1,110	1,110	1,110	**0**	**0**	**0**	**0**	**0**
	t_m	**0.4**	0.8	2	5,723	367	1,240	1,381	10
	n_m	1,110	1,110	1,110	**0**	**0**	**0**	**0**	**0**
BDD	t	30	**10**	19	924	**3**	18	24	mem
	n	19,139	19,012	19,012	**11**	**11**	**11**	**11**	-
	t_m	30	11	20	634	1	6	9	mem
	n_m	20,956	20,587	20,587	**11**	**11**	**11**	**11**	-
Dubois	t	2,026	1,432	906	1,413	**807**	1,005	6,751	1,084
	n	1,008M	1,070M	1,070M	419M	**401M**	**401M**	**401M**	**401M**
	t_m	673	481	318	563	**264**	369	2,372	371
	n_m	359M	373M	373M	156M	141M	141M	141M	141M
Aim-100	t	6,390	136	**70**	748	1,019	863	3,899	674
	n	643M	18M	18M	**10M**	34M	34M	34M	32M
	t_m	1,435	97	50	0.8	22	14	94	**0.2**
	n_m	103M	13M	13M	1,131	875,979	875,985	875,985	**1,522**
Aim-200	t	13	11	5	**3**	4	**3**	15	4
	n	479,073	846,357	846,357	**14,937**	97,529	104,748	88,541	75,209
	t_m	2.7	2.5	1.5	0.6	0.6	**0.4**	1.3	0.6
	n_m	159,591	120,897	120,897	**937**	2,887	2,887	3,236	4,276

The results are more varied on the *Aim* classes where the winning algorithm is not clear. On most instances there are (often huge) differences in favor of our methods compared to Abscon. However, there are cases where Abscon visits considerably fewer nodes than our algorithms, resulting in much better run times. This is contrary to what is expected since a stronger local consistency should typically result in a smaller tree size. However, the interaction with the variable ordering heuristic is sometimes

unpredictable, given also the different implementations of Abscon and our solver. For example, if we look at the results from aim-100-2-0-sat-4 we can see a huge difference in node visits between Abscon and our implementation of STR2. This is because, despite both applying GAC, the two solvers follow quite different search paths. Consequently, the comparison of our methods on *Aim* is more meaningful against our own implementation of STR2. In this case, our methods typically outperform STR2 by very large margins.

Looking at the mean and median performances in Table 3, maxRPWC+r (and HOSTR2^h) is faster than STR2 in the *BDD*, *Dubois* and *Aim* classes with the differences being considerable in the two *Aim* classes. On the other hand, STR2 dominates in the *Random* and *Positive table* classes, with the differences being very significant in the latter. Median CPU times follow the average CPU times in the majority of the classes, but they are more representative on *Aim* where results are diverse, since there are instances solved very quickly and instances solved in thousands of seconds.

Regarding the pruning power of the strong local consistency algorithms, which is to some extent reflected on node visits, it is worth noticing that the differences between the strong consistency methods are negligible in the majority of the classes. The *Aim* classes are exceptions as maxRPWC1 visits significantly less nodes compared to our algorithms, and this results in competitive CPU times. Importantly, the very small differences in node visits between maxRPWC1 and our algorithms on problems with constraints of large arity (*Positive table* and *BDD*) mean that the very large differences in run times can only be explained by the algorithmic optimizations that we propose in this paper.

5.2.1 Pairwise comparisons

CPU times from all tested instances comparing maxRPWC+r to HOSTR2^h, maxRPWC1, STR2, and STR2_abs are presented in Figures 4, 5, 6, 8, respectively, in a logarithmic scale. Different signs display instances from different problem classes and are calculated by CPU time ratios of the compared algorithms. Points placed above the diagonal correspond to instances that were solved faster by maxRPWC+r.

In Figure 4 most instances are gathered around the diagonal indicating closely matched performance between maxRPWC+r and HOSTR2^h. maxRPWC+r is superior to HOSTR2^h, as detailed above, on *Positive table-8* and inferior on *BDD* and *Aim-100*, but without significant differences.

Comparing maxRPWC+r to maxRPWC1 in Figure 5, we can see the benefits of our approach: Only a few instancesfrom *BDD* and *Aim* are below the diagonal indicating that maxRPWC+r is clearly superior to maxRPWC1. Also, there are many instances where maxRPWC1 thrashes while maxRPWC+r does not.

Looking at Figure 6 we see a more varied picture. Although on most instances maxRPWC+r and STR2 are closely matched, there are numerous instances where one of the two methods thrashes and vice versa. This demonstrates that efficient algorithms for strong local consistencies on table constraints constitute a useful addition to the standard GAC approach, but at the same time further research is required to develop methods that can be more robust than state-of-the-art GAC algorithms such as STR2 on a wider range of problems.

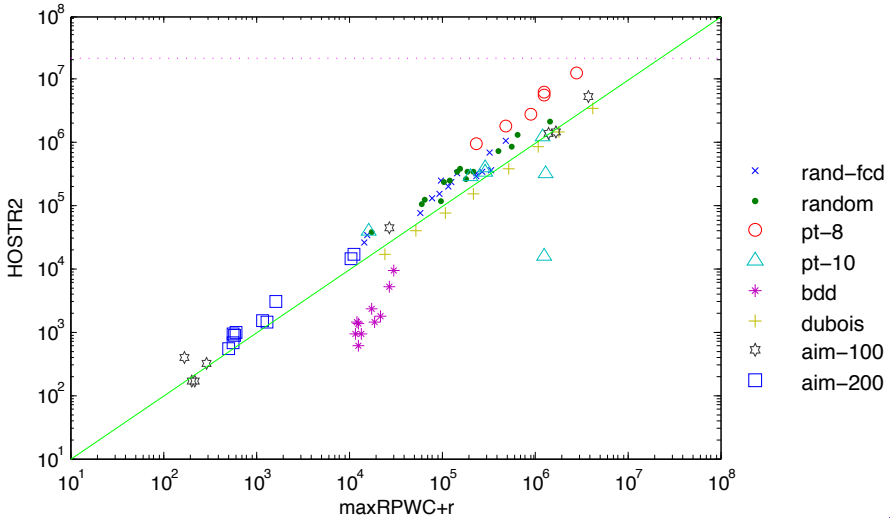

Figure 4: HOSTR2^h vs. maxRPWC+r.

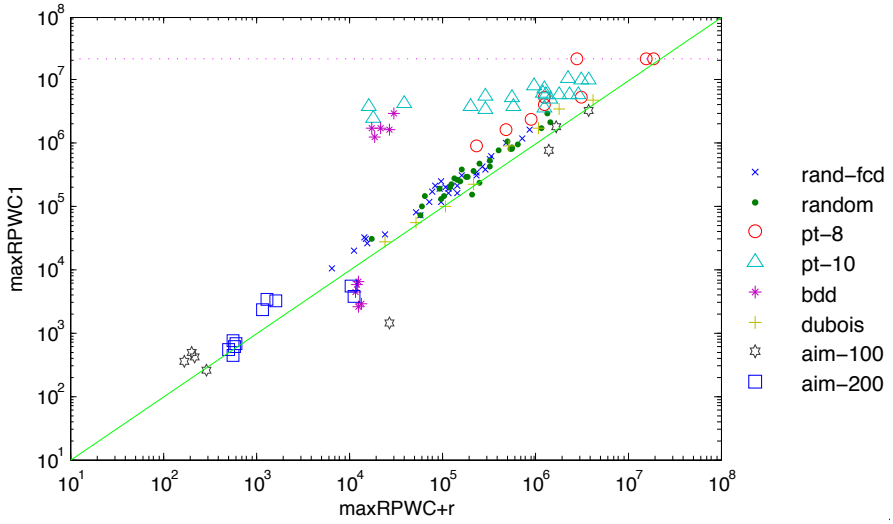

Figure 5: maxRPWC1 vs. maxRPWC+r.

A similar pattern emerges when comparing HOSTR2^h to STR2. This is shown in Figure 7. Specifically, HOSTR2^h is superior on instances with constraints of small arity but many constraint intersections, like *Dubois* and *Aim*. On the other hand, despite being inferior on many instances from both the *Positive table* classes, it is faster by an order of magnitude on *BDD* instances where the intersections between constraints include up to 16 variables.

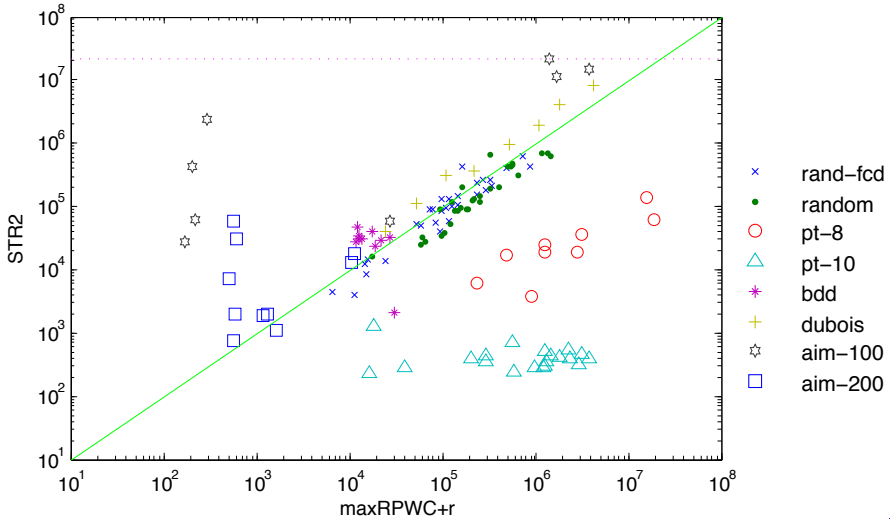

Figure 6: STR2 vs. maxRPWC+r.

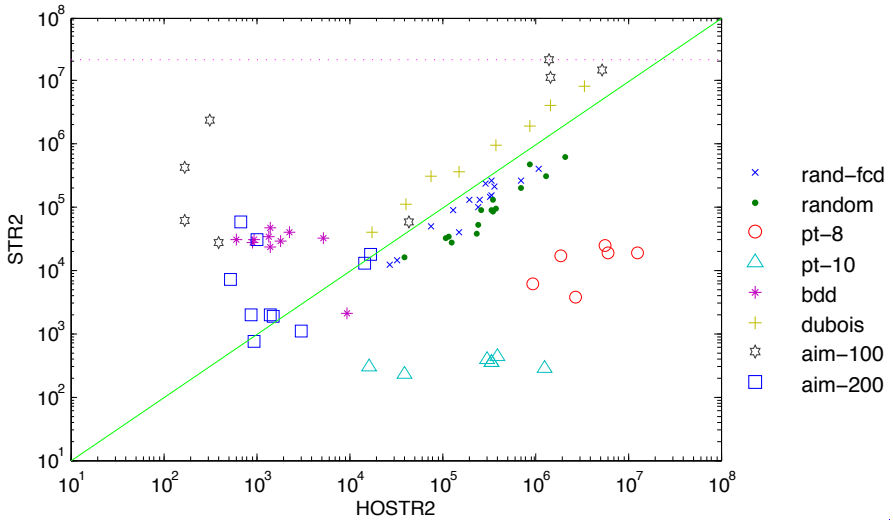

Figure 7: STR2 vs. HOSTR2^h.

In Figure 8 we compare STR2_abs to maxRPWC+r. Results from a comparison between GAC-va_abs and maxRPWC+r are very similar and thus we do not present the corresonding figure. Although Figure 8 displays significant differences in favor of STR2_abs on some instances, there are still quite a few instances where maxRPWC+r is superior, often by large margins. This shows the potential of strong local consistencies on table constraints. As discussed, we believe that integrating an algorithm like maxRPWC+r in Abscon, or some other state-of-the-art solver, would result in much better run time performance.

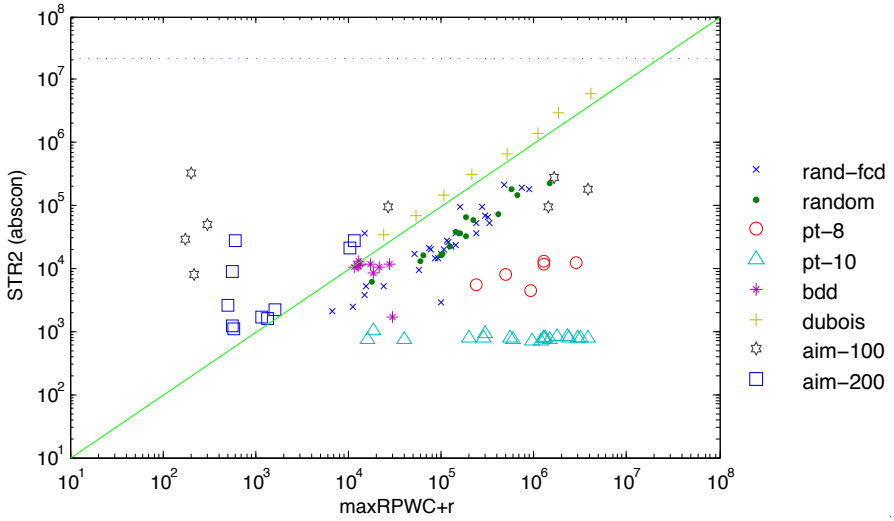

Figure 8: `STR2_abs` vs. `maxRPWC+r`.

5.2.2 Discussion

We now draw some general conclusions correlating the performance of the proposed algorithms and the characteristics of the tested problems, based on the experimental results.

First, it is evident that the performance of our methods is highly dependant on the presence of non-trivial constraint intersections as their exploitation offers extra pruning compared to GAC algorithms. The absense of non-trivial intersections makes these methods obsolete.

Second, the arity of the constraints, the number of shared variables between constraints, and the size of the tables are important factors when comparing our algorithms to existing maxRPWC algorithms. This is because for large arity constraints the improvements in saving constraint checks and avoiding redundant operations on data structures become more evident. Results from the classes that include constraints of large arity (*Positive table*, *BDD*) confirm this.

However, the above factors do not appear to be decisive when comparing our methods to GAC algorithms. There are problems with small arities as well as others with large arities where our methods are better (resp. worse) than GAC algorithms, i.e., the last (resp. the first) four classes of Table 3. The same holds when considering the size of the tables. Also, as explained, the number of shared variables is a decisive factor when comparing against `eSTR` because this algorithm may run out of memory when this number is high.

Third, the domain size of the variables seems to be an important factor when comparing our methods to GAC algorithms or `eSTR`. Our methods are mostly competitive on problems with small domains (*BDD*, *Dubois*, *Aim*). This is because all of our methods include, to some extent, an interleaved exploration of valid and allowed tuples. For problems with large domain sizes the number of valid tuples can be quite high, at least

near the top of the search tree when only a few value deletions have been made. This can slow down the interleaved search for supports significantly. `eSTR` does not have this problem because it employs quite different reasoning.

6 Related Work

GAC algorithms for table constraints have attracted considerable interest dating back to GAC-Schema [4]. This generic method can be instantiated to either a method that searches the lists of allowed tuples for supports, or to one that searches the valid tuples. The `GAC-va` algorithm improves on GAC-Schema by interleaving the exploration of allowed and valid tuples using binary search [23]. The interleaved exploration of allowed and valid tuples is also the main idea in [26]. However, in this case it is implemented through the use of an elaborate data structure (*Hologram*) introduced by Lhomme [25]. Recent algorithms are partly based on similar ideas as the Hologram method [29]. That is, they hold information about removed values in the propagation queue and utilize it to speed up support search.

Alternative data structures for storing and handling table constraints have been also introduced [11]. In that paper, a Trie structure was the most efficient among the ones tested. Katsirelos and Walsh used a compact representation for allowed and disallowed tuples which can be constructed from a decision tree that represents the original tuples [17].

Simple Tabular Reduction (`STR`) [34] and its variants, `STR2` [18] and `STR3` [22], constitute an alternative and efficient approach to enforcing GAC based on the dynamic maintenance of the support tables. Finally, multi-valued decision diagrams have been used to store and process table constraints [9].

Experimental results show that the most competitive approaches are the ones based on `STR` and the `MDD` approach. The algorithm of [23], which `maxRPWC+` extends, is very competitive with the Trie approach, outperforms the Hologram method and has the advantage of easier implementation and lack of complex data structures over all other methods. Albeit, it is clearly slower than the best methods on most problems.

With respect to strong local consistencies, there is considerable older work on relation filtering consistencies. Such methods take advantage of the intersections between constraints in order to identify and remove inconsistent tuples or to add new constraints to the problem (e.g., [35, 13]). Quite recently, strong domain filtering consistencies have received attention [7, 32]. Moreover, efficient ways to apply relational consistencies were proposed and new consistencies of this type were introduced for binary constraints [19, 14, 38] and non-binary constraints [16, 37].

The works of Woodward et. al. concern the application of various relational consistencies on (mainly) table constraints through the exploitation of a problem's dual encoding [16, 37]. Experimental results show that very high local consistencies (higher than FPWC) can pay off when they are applied through efficient algorithms. However, the proposed algorithms were not compared to state-of-the-art GAC methods such as `STR2`. Interestingly, the so called index-tree data structure was proposed to efficiently locate all tuples in a constraint's relation that are consistent with a tuple of another intersecting constraint [16]. This is a contribution that is orthogonal to ours and it would

be interesting to use this data structure in the context of our algorithms.

The idea of avoiding checks for PW-support loss that algorithms maxRPWC+ and maxRPWC+r employ is inspired by a similar idea concerning maxRPC, a related local consistency property for binary constraints [36, 1, 2]. Although the restricted versions of maxRPC algorithms (called light maxRPC [36]) achieve only an approximation of maxRPC, they are much faster compared to full maxRPC algorithms, and they very close in terms of pruning. In addition, the idea of making a lighter use of the $Last$ structure by not maintaining it during search, which is employed by maxRPWC+r, is inspired by the residue-based AC3^{rm} [21] and maxRPC3^{rm} algorithms [2].

Furthermore, both fHOSTR and eSTR [24] achieve the same level of consistency, namely GAC+PWC. Their difference lies in the different data structures they utilize which results in a different handling of PW-checks in their respective *isPWconsistent* function. Whenever a tuple $\tau \in rel[c]$ is verified as valid, both algorithms iterate over each constraint c_k that intersects with c to check if there exists a PW-support for τ in $rel[c_k]$. In eSTR, this is done through a look-up in the appropriate counter stored for each intersection between any two constraints. At any time each counter holds the number of valid tuples in c_k's relation table that include a specific combination of values for the set of variables that are common to both c and c_k. Importantly, this check is done in constant time, while in fHOSTR we need to search between valid and allowed tuples (for the case of function 6). However, experiments in this paper showed that there are cases where while strong consistency methods outperform GAC, the use of HOSTR or maxRPWC+r is inevitable (i.e., in problems with large constraint intersections) because eSTR exhausts the memory resources.

Importantly, strong local consistency techniques for table constraints have recently started to find their way into state-of-the-art CP solvers such as Abscon[5]. The latest version of Abscon offers an unpublished algorithm which achieves FPWC and can be regarded as a variant of eSTR^{w} [24]. This development demonstrates that strong local consistency methods, such as the ones presented in this paper, are compatible with CP solvers from a software engineering point of view, and can used to extend the solvers' arsenal of techniques for handling table constraints.

Finally, a simple and efficient way to incorporate strong local consistency reasoning into CP solvers, through the use of a reformulation, was very recently proposed [27]. The main idea of the technique, called factor encoding, is to formulate new variables by extracting commonly shared variables from the constraints' scopes and then to reattach them back to the constraints where they come from. The reformulation transforms a given problem into another equivalent one, and by enforcing GAC on the new problem, a quite stronger consistency property is achieved on the original problem. This approach was compared experimentally to the FPWC algorithm of Abscon with favourable results.

7 Conclusion

We presented specialized algorithms for table constraints that achieve local consistencies stronger than the standard GAC. These algorithms build on and extend existing

[5] http://www.cril.univ-artois.fr/∼lecoutre/software.html

algorithms for GAC and maxRWPC and contribute to both directions of domain and relation filtering local consistencies. Experimental results demonstrated the usefulness of the proposed algorithms in the presence of intersecting table constraints, showing that the best among them can be competitive with a state-of-the art GAC algorithm (`STR2`) and sometimes even with the state-of-the-art CP solver Abscon.

In some cases (e.g., *Aim* classes) there can be huge differences in the numbers of search tree nodes which are reflected on important CPU time gains, even compared to a highly optimized solver like Abscon. However on other problems the extra processing performed by our methods slows down the solver, sometimes considerably (e.g the *Positive table* classes).

Our paper binds together recent advances on GAC for table constraints and strong local consistencies contributing to both directions. Specifically, we offer efficient methods for strong filtering in cases of intersecting table constraints, and we make strong consistencies more practical by moving from generic to specialized algorithms.

We believe that the presented work can pave the way for the design and implementation of even more efficient strong consistency methods for table constraints. Also, it can perhaps help initiate a wider study on specialized strong consistency algorithms for specialized (global) constraints.

8 Acknowledgements

We owe a great thanks to Christophe Lecoutre for giving us the executable of the Abscon solver. We also thank the anonymous reviewers for helping us improve the quality of this paper.

References

[1] T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. Improving the performance of maxRPC. In *Proceedings of CP'10*, pages 69–83, 2010.

[2] T. Balafoutis, A. Paparrizou, K. Stergiou, and T. Walsh. New algorithms for max restricted path consistency. *Constraints*, 16(4):372–406, 2011.

[3] T Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in arc consistency algorithms. In *ECAI-08 Workshop on Modeling and Solving Problems with Constraints*, 2008.

[4] C. Bessiere and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1996.

[5] C. Bessiere and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In *Proceedings of CP'96*, pages 61–75, Cambridge MA, 1996.

[6] C. Bessiere, J.C. Régin, R. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[7] C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.

[8] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, Valencia, Spain, 2004.

[9] K. Cheng and R. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[10] R. Debruyne and C. Bessiere. Domain filtering consistencies. *JAIR*, 14:205–230, 2001.

[11] I. P. Gent, C. Jefferson, I. Miguel, and Nightingale P. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of AAAI'07*, pages 191–197, 2007.

[12] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint satisfaction problems: Achieving pairwise consistency using an associated binary representation. In *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[13] P. Jégou. On the consistency of general constraint satisfaction problems. In *Proceedings of AAAI'93*, pages 114–119, 1993.

[14] P. Jégou and C. Terrioux. A new filtering based on decomposition of constraint sub-networks. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 263–270, 2010.

[15] U. Junker. Preference-based problem solving for constraint programming. In *CSCLP*, pages 109–126, 2007.

[16] S. Karakashian, R. Woodward, C. Reeson, B. Choueiry, and C. Bessiere. A first practical algorithm for high levels of relational consistency. In *Proceedings of AAAI'10*, pages 101–107, 2010.

[17] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393. Springer-Verlag, 2007.

[18] C. Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[19] C. Lecoutre, S. Cardon, and J. Vion. Conservative dual consistency. In *Proceedings of AAAI'07*, pages 237–242, 2007.

[20] C. Lecoutre, S. Cardon, and J. Vion. Second-order consistencies. *J. Artif. Int. Res.*, 40(1):175–219, 2011.

[21] C. Lecoutre and F. Hemery. A study of residual supports in arc cosistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

[22] C. Lecoutre, C. Likitvivatanavong, and R. H. C. Yap. A path-optimal GAC algorithm for table constraints. In *ECAI*, pages 510–515, 2012.

[23] C. Lecoutre and R. Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP'06*, pages 284–298, 2006.

[24] Christophe Lecoutre, Anastasia Paparrizou, and Kostas Stergiou. Extending STR to a higher-order consistency. In *Proceedings of AAAI'13*, pages 576–582, 2013.

[25] O. Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *Proceedings of CPAIOR'04*, pages 209–224, 2004.

[26] O. Lhomme and J.C. Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of AAAI'05*, pages 405–410, 2005.

[27] C. Likitvivatanavong, W. Xia, and R. H. C. Yap. Higher-order consistencies through GAC on factor variables. In *Proceedings of CP'14*, pages 497–513, 2014.

[28] A. Mackworth. On reading sketch maps. In *Proceedings of IJCAI'77*, pages 598–606, Cambridge MA, 1977.

[29] J.B. Mairy, P. Hentenryck, and Y. Deville. An optimal filtering algorithm for table constraints. In *CP*, pages 496–511, 2012.

[30] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[31] A. Paparrizou and K. Stergiou. An efficient higher-order consistency algorithm for table constraints. In *Proceedings of AAAI'12*, pages 535–541, 2012.

[32] K. Stergiou. Strong inverse consistencies for non-binary CSPs. In *Proceedings of ICTAI'07*, pages 215–222, 2007.

[33] K. Stergiou and T. Walsh. Inverse Consistencies for non-binary constraints. In *Proceedings of ECAI'06*, pages 153–157, 2006.

[34] J. R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007.

[35] P. van Beek and R. Dechter. On the minimality and global consistency of row-convex constraint networks. *JACM*, 42(3):543–561, 1995.

[36] J. Vion and R. Debruyne. Light algorithms for maintaining max-RPC during search. In *Proceedings of SARA'09*, 2009.

[37] R. J. Woodward, S. Karakashian, B. Y. Choueiry, and C. Bessiere. Solving difficult CSPs with relational neighborhood inverse consistency. In *Proceedings of AAAI'11*, pages 112–119, 2011.

[38] R. J. Woodward, S. Karakashian, B. Y. Choueiry, and C. Bessiere. Revisiting neighborhood inverse consistency on binary CSPs. In *Proceedings of CP'12*, pages 688–703, 2012.