An Efficient Higher-Order Consistency Algorithm for Table Constraints *

Anastasia Paparrizou and **Kostas Stergiou**
Department of Informatics and Telecommunications Engineering
University of Western Macedonia, Greece

Abstract

Table constraints are very 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 an efficient algorithm for table constraints that achieves a stronger local consistency than GAC. This algorithm, called maxRPWC+, is based on the local consistency maxRPWC and allows the efficient handling of intersecting table constraints. Experimental results from benchmark problems demonstrate that maxRPWC+ is clearly more robust than a state-of-the-art GAC algorithm in classes of problems with interleaved table constraints, being orders of magnitude faster in some of these classes.

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. Given their importance in CP, it is not surprising that numerous specialized algorithms that achieve GAC (i.e. domain consistency) on table constraints have been proposed. Since GAC is a property defined on single constraints, algorithms for GAC operate on one constraint at a time trying to filter infeasible values from the variables of the constraint.

Higher-order consistencies that are stronger than GAC have been proposed in the literature but are not widely used. Typically, such local consistencies take advantage of intersections between constraints to remove inconsistent tuples or to add new constraints. Recently there has been renewed interest for higher-order consistencies as new ones have been proposed (Lecoutre, Cardon, and Vion 2007; Bessiere, Stergiou, and Walsh 2008) or efficient algorithms for existing ones have been devised (Karakashian et al. 2010). Interestingly, the local consistencies introduced in (Bessiere, Stergiou, and Walsh 2008) are domain filtering, meaning that they only prune values from the domains of variables and therefore do not alter the structure of the constraint hypergraph or the constraints' relations. This makes them easier to integrate into existing solvers. One of the most promising consistencies of this type is Max Restricted Pairwise Consistency (maxRPWC) (Bessiere, Stergiou, and Walsh 2008).

In this paper we propose a specialized algorithm that is based on maxRPWC and allows the efficient handling of intersecting table constraints. The algorithm, called maxRPWC+, extends the state-of-the-art GAC algorithm for table constraints given in (Lecoutre and Szymanek 2006) and specializes the generic maxRWPC algorithm maxRPWC1. The proposed algorithm incorporates several techniques that help alleviate redundancies displayed by existing maxRPWC algorithms. Specifically, we consider a simple approximation of maxRPWC which achieves slightly less pruning than maxRPWC, at considerably reduced cost. Then we apply two complementary methods that can speed up the process of checking for consistency through the exploitation of a simple data structure already used in existing maxRPWC and GAC algorithms. The first (resp. second) method targets cases where a tuple is consistent (resp. is inconsistent) and tries to quickly verify this. maxRPWC+, achieves a local consistency level stronger than GAC and incomparable to maxRPWC.

Although maxRPWC+ is specialized for table constraints, we show that it can be also applied on intensional constraints. This may be useful in cases of constraints without specialized filtering algorithms, or to simply explore the potential of a higher-order consistency on any given constraint.

Experimental results from benchmark problems demonstrate that maxRPWC+ is clearly more robust than a state-of-the-art GAC algorithm in classes of problems with interleaved table constraints, being orders of magnitude faster in some of these classes. In addition, maxRPWC+ is considerably faster than generic maxRPWC algorithms, showing that specialized algorithms for higher-order consistencies can be very useful in practice.

*This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Heracleitus II. Investing in knowledge society through the European Social Fund.

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, and $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e constraints, with maximum arity k. Each constraint c is a pair $(var(c), rel(c))$, where $var(c) = \{x_1, \ldots, x_m\}$ is an ordered subset of $\mathcal{X}$ referred to as the *constraint scope*, and $rel(c)$ is a subset of the *Cartesian* product $D(x_1) \times \ldots \times D(x_m)$ that specifies the allowed combinations of values for the variables in $var(c)$.

Each tuple $\tau \in rel(c)$ is an ordered list of values $< (x_1, a_1), \ldots, (x_m, a_m) >$ s.t. $a_j \in D(x_j)$, $j = 1, \ldots, m$. Given a constraint c, a variable $x_i \in var(c)$, and a tuple $\tau \in rel(c)$, we denote by $\tau[x_i]$ the projection of τ on x_i. A tuple is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable. For any constraint c we denote by $\top$ (resp. $\bot$) a dummy tuple s.t. $\tau < \top$ (resp. $\tau > \bot$) for any tuple $\tau \in rel(c)$. Given two constraints c_i and c_j, if $var(c_i) \cap var(c_j) \neq \emptyset$ we say that the constraints *intersect*. We denote by f_{max} (resp. f_{min}) the maximum (resp. minimum) number of variables that are common to any two constraints that intersect on more than one variable.

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. $x_i \in var(c)$, there exists a valid tuple $\tau \in rel(c)$ s.t. $\tau[x_i] = a_i$. 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.

Several alternative consistencies have been proposed with domain filtering consistencies being especially interesting. Examples of such consistencies include SAC (Debruyne and Bessière 2001), RPWC and maxRPWC (Bessiere, Stergiou, and Walsh 2008), rPIC (van Beek and Dechter 1995). A theoretical and experimental evaluation presented in (Bessiere, Stergiou, and Walsh 2008) demonstrated that maxRPWC is a promising alternative to GAC.

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

Following (Debruyne and Bessière 2001) 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 were proposed in (Bessiere, Stergiou, and Walsh 2008). 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 f_{max}. 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 is 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.

maxRPWC+

Table constraints are very common in applications from configuration and databases among others. As a result, in the last few years many specialized GAC algorithms for table constraints have been proposed (Lhomme and Régin 2005; Lecoutre and Szymanek 2006; Gent et al. 2007; Cheng and Yap 2010; Lecoutre 2011). However, these algorithms cannot exploit possible intersections that may exist between different constraints. On the other hand, existing algorithms for maxRPWC and other related higher-order consistencies are generic and thus very expensive. Hence, any extra pruning that they may achieve by exploiting intersections will be outweighed by the cpu time overhead.

We will now describe a specialized algorithm for table constraints that is based on maxRPWC and builds upon the generic maxRPWC algorithm `maxRPWC1` and the specialized GAC algorithm of (Lecoutre and Szymanek 2006) (called `GAC-va` hereafter). The presented algorithm, called `maxRPWC+`, not only specializes maxRPWC to table constraints, but also introduces several techniques that help eliminate redundancies displayed by existing algorithms. As in `GAC-va`, the main idea behind `maxRPWC+` is to interleave support and validity checks.

Restricted maxRPWC 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. 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. A similar approximation of the related binary local consistency maxRPC has also been shown to be efficient compared to full maxRPC (Vion and Debruyne 2009; Balafoutis et al. 2011).

Algorithm `maxRPWC+` uses the following data structures:

- For each variable-value pair (x_i, a_i) and each constraint c involving x_i, $allowed(c, x_i, a_i)$ 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 $x_i \in var(c)$, $LastPWC_{c,x_i,a_i}$ gives the most recently discovered (and thus lexicographically smallest) maxRPWC-support of a_i in c.

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

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);
```

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 GAC-va. This function first checks if $LastPWC_{c_i,x_j,a_j}$, which is the most recently found maxRPWC-support, and thus also GAC-support, is still valid. If this is true, $\tau = LastPWC_{c_i,x_j,a_j}$ is returned, else the valid and allowed tuples of c_i are visited in an alternating fashion to locate the lexicographically smallest valid and allowed tuple τ of c_i, such that $\tau > LastPWC_{c_i,x_j,a_j}$ and $\tau[x_j] = a_j$. 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 2 *isPWconsistent+* (c_i, τ): **boolean**

```
1: for each c_k ≠ c_i s.t. |var(c_k) ∩ var(c_i)| > 1 do
2:     PW=FALSE;
3:     max_τ' ← ⊥;
4:     for each x_k ∈ var(c_k) ∩ var(c_i) do
5:         τ' ← LastPWC_{c_k,x_k,τ[x_k]};
6:         if isValid(c_k, τ') AND τ'[var(c_k) ∩ var(c_i)] = τ[var(c_k) ∩
           var(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;
```

The process of checking if a tuple τ of a constraint c_i is PW consistent involves iterating over each c_k that intersects with c_i on at least two variables[1] 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 $LastPWC$ data structure as we now explain.

[1]On constraints that intersect on one variable maxRPWC is equivalent to GAC (Bessiere, Stergiou, and Walsh 2008).

Fast check for PW-support For each variable x_k belonging to the intersection of c_i and c_k, we check if $\tau' = LastPWC_{c_k,x_k,\tau[x_k]}$ 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'.

Else, we find max_τ' the lexicographically greatest $LastPWC_{c_k,x_k,\tau[x_k]}$ 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.

Fast check for lack of PW-support 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 cannnot 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 $max\_\tau'[var(c_k) \cap var(c_i)] = \tau[var(c_k) \cap var(c_i)]$. 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). 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 $\top$.

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 $var(c_1) = \{x_1, x_2, x_3, x_4\}$ and $var(c_2) = \{x_3, x_4, x_5, x_6\}$. 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 $LastPWC_{c_2,x_3,2}$ and $LastPWC_{c_2,x_4,1}$ 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 *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 $max\_\tau' =< (x_1, a_1), ..., (x_m, a_m) >$ then this tuple is scanned from left to right. If the currently examined variable x_k belongs to $var(c_k) \cap var(c_i)$ 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 $var(c_k) \cap var(c_i)$ then if the value it takes in max_τ' is the last value in its domain, we continue scanning (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*.

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

1: **if** $\neg$ *isValid*(c_k, max_τ') **then** $max\_\tau' \leftarrow$ *setNextTuple*$(c_i, \tau, c_k, max\_\tau')$;
2: **if** $max\_\tau' \neq \top$ **then** $\tau' \leftarrow$ *checkPWtuple*$(c_i, \tau, c_k, max\_\tau')$;
3: **else return** $\top$;
4: **while** $\tau' \neq \top$ **do**
5: $\quad \tau'' \leftarrow$ *binarySearch*$(allowed(c_k, x_{ch}, \tau'[x_{ch}]), \tau')$;
6: $\quad$ **if** $\tau'' = \tau'$ OR *isValid*(c_k, τ'') **then return** τ'';
7: $\quad$ **if** $\tau'' = \top$ **then return** $\top$;
8: $\quad \tau' \leftarrow$ *setNextTuple*(c_i, τ, c_k, τ'');
9: **return** $\top$;

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.

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

1: **for each** $x_k \in var(c_k)$ **do**
2: $\quad$ **if** $x_k \notin var(c_k) \cap var(c_i)$ **then**
3: $\quad\quad$ **if** $max\_\tau'[x_k]$ is last value in $D(x_k)$ **then continue**;
4: $\quad\quad$ **else break**;
5: $\quad$ **else**
6: $\quad\quad$ **if** $max\_\tau'[x_k] < \tau[x_k]$ **then break**;
7: $\quad\quad$ **if** $max\_\tau'[x_k] > \tau[x_k]$ **then return** $\top$;
8: **return** max_τ';

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 $var(c_k)$ that is lexicographically greater than τ'', such that $\tau'[var(c_k) \cap var(c_i)] = \tau[var(c_k) \cap var(c_i)]$ (line 8). If *setNextTuple* returns $\top$ the search terminates, otherwise, we continue to check if the returned tuple is allowed as explained above, and so on.

Selecting the list of allowed tuples Since there are $|var(c_k) \cap var(c_i)|$ 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 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 $\top$.
3. Select the leftmost variable in $var(c_k) \cap var(c_i)$.
4. Select the rightmost variable in $var(c_k) \cap var(c_i)$.

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, $var(c_k) \cap var(c_i) = \{x_1, x_4\}$, $\tau[x_1] = 1$, $\tau[x_4] = 0$, 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 $allowed(c_k, x_4, 0)$.

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 $S = |allowed(c_{ch}, x_{ch}, \tau'[x_{ch}])|$. 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 t belonging to such a sequence $t[var(c_k) \cap var(c_i)] = \tau[var(c_k) \cap var(c_i)]$. 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 maximun 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($d + k.log(S)$) (Lecoutre and Szymanek 2006). Therefore, for d values the complexity of *revisePW+* is O($d.N(e.M(d + k.log(S)) + (d + k.log(S))$)=O($d.e.N.M(d + k.log(S))$).□

If *isPWconsistent+* is embedded within an AC3-like algorithm (as `maxRPWC1` is) then the worst-case time complexity of `maxRPWC+` will be O($e^2.k.d.N.M(d + k.log(S))$). Assuming the implementation of (Lecoutre and Szymanek 2006), 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 $LastPWC$ structure.

Regarding the pruning power of `maxRPWC+`, it is easy to show that it is strictly stronger than GAC and incomparable to maxRPWC. With respect to the latter, a maxRPWC algorithm may achieve stronger pruning than `maxRPWC+` because it detects PW-support loss in addition to maxRPWC-support loss. On the other hand, the check for lack of PW-support enables `maxRPWC+` to prune extra values compared to maxRPWC.

maxRPWC+ for Intensional Constraints

Although `maxRPWC+` is specialized for table constraints, it can be applied on intensional constraints after some modifications. This may be useful in cases of constraints without specialized filtering algorithms, or to simply explore the potential of a higher-order consistency on any given constraint without having to invent specialized algorithms.

Function 5 *seekPWsupport-v* ($c_i, \tau, c_k, max\_\tau'$)

1: **if** $\neg$ *isValid*(c_k, max_τ') **then** $max\_\tau' \leftarrow$ *setNextTuple*($c_i, \tau, c_k, max\_\tau'$);
2: **if** $max\_\tau' \neq \top$ **then** $\tau' \leftarrow$ *checkPWtuple*($c_i, \tau, c_k, max\_\tau'$);
3: **else return** $\top$;
4: **while** $\tau' \neq \top$ **do**
5: **if** *isConsistent*(c_k, τ') **then return** τ';
6: $\tau' \leftarrow$ *setNextTuple*(c_i, τ, c_k, τ');
7: **return** $\top$;

Function *seekSupport-v* for intensional constraints is similar to the corresponding function (*seekSupport*) for extensional ones. The difference is that the search for GAC-support is a linear scan of the tuples after $LastPWC_{c_i,x_j,a_j}$ in lexicographical order. That is, only the list of valid tuples is traversed.

The corresponding function that searches for a PW-support is Function *seekPWsupport-v* (Function 5). Lines 5-6 show the different approach we use on intensional constaints. Instead of interchangeably visiting valid and allowed tuples, we just check if the valid tuple τ' satisfies the constraint c_k by calling function *isConsistent*. If τ' is inconsistent then a new tuple τ' is constructed by *setNextTuple* (line 6), such that $\tau'[var(c_k) \cap var(c_i)] = \tau[var(c_k) \cap var(c_i)]$. If *setNextTuple* returns $\top$ the search terminates, otherwise, we continue to check if the returned tuple is consistent as explained above, and so on.

The worst-case time complexity of `maxRPWC+` for intensional constraints is O($e^2k^2d^{2k-f_{min}}$), which is the same as that of `maxRPWC1` if we consider that in the worst case $p = 2k - f_{min}$. The space complexity of `maxRPWC+` is O(ekd) which is the space required for $LastPWC$ and is the same as `maxRPWC1` but lower than both `maxRPWC2` and `maxRPWC3`.

Experiments

We ran experiments on benchmark non-binary problems from the CSP Solver Competition[2]. The arities of the constraints in these problems range from 3 to 18. We tried the following classes: *random problems*, *forced random problems*, *chessboard coloration*, *Schurr's lemma*, *aim*, *modified Renault*, *positive table constraints* and *BDD*. Some other classes typically used in the evaluation of GAC algorithms for table constraints, such as *crossword puzzles* and *travelling salesman*, were ommitted because they include constraint intersections on at most one variable. As explained, `maxRPWC+` cannot achieve extra filtering compared to GAC in such problems.

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. Search used a binary branching scheme, the *dom/wdeg* heuristic for variable ordering and lexicographical value ordering heuristic (Boussemart et al. 2004).

In Table 1 we present indicative results from search algorithms that apply `GAC-va`, `maxRPWC1` and `maxRPWC+` on various instances, as well their average performance in each problem class. We also include results from the restricted version of `maxRPWC1`, denoted `RmaxRPWC1`, which, as demonstrated, is almost always beneficial compared to the full version. `maxRPWC+` can clearly outperform `GAC-va`, even by orders of magnitude, on many problem classes (i.e. *Positive table-10*, *aim*, *BDD*). Specifically, `GAC-va` reached the cutoff limit on *Positive table-10* instances due to the high memory consumption. In addition, `maxRPWC+` is considerably faster than the generic maxRPWC algorithm, showing that specialized algorithms for higher-order consistencies can be very useful in practice.

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

Specifically, maxRPWC+ is always faster than RmaxRPWC1 and can be orders of magnitude faster than maxRPWC1. There are classes (i.e. *Positive table-8*, *Renault-modified*), where maxRPWC1 was not able to solve the majority of the instances within 6 hours (our cutoff limit). Similarly, GAC-va could not solve any of the *Positive table-10* instances within 6 hours. Significantly, maxRPWC+ solved all instances within the time limit. There are problems where GAC-va was faster than maxRPWC+ (e.g. *Positive table-8*), but the differences were never very significant.

Table 1: Cpu times (t) in secs and nodes (n) from various representative problem instances.

Instance		GAC-va	maxRPWC1	RmaxRPWC1	maxRPWC+
rand-3-20-20-	t	147	95	283	**46**
60-632-fcd-2	n	94,424	8,165	46,192	9,973
rand-3-20-20-	t	108	292	22	**16**
60-632-fcd-15	n	85,940	39,972	5,790	5,800
Rand-fcd	t	136	400	229	**135**
AVERAGE	n	105,852	48,559	51,006	42.241
rand-3-20-20	t	122	144	44	**29**
-60-632-8	n	105,182	17,349	9,060	8,447
rand-3-20-20	t	**101**	521	230	165
-60-632-9	n	73,408	52,311	50,818	50,353
Random	t	**304**	786	460	323
AVERAGE	n	227,085	86,863	102,437	98,800
pt-8-20-5-	t	**1,203**	-	3,572	1,266
18-80-4	n	37,466	-	10,654	9,199
pt-8-20-5-	t	**493**	-	1,741	564
18-80-7	n	15,845	-	4,423	4,126
Positive table-8	t	**1,042**	-	4,650	1,641
AVERAGE	n	47,073	-	15,142	14,349
pt-10-20-10-	t	-	2,302	2,393	**13**
5-10000-2	n	-	0	0	0
pt-10-20-10-	t	-	3,723	3,861	**690**
5-10000-4	n	-	0	0	0
Positive table-10	t	-	3,864	4,013	**653**
AVERAGE	n	-	0	0	0
aim-200-1-	t	85	**23**	94	61
6-unsat-2	n	680,774	154,540	708,211	461,185
aim-200-2-	t	705	**0.8**	1.5	1.3
0-sat-4	n	4,180,497	1,060	3,882	3,882
aim	t	165	**6**	33	30
AVERAGE	n	1,038,481	32,038	238,816	214,571
bdd-21-133-	t	5,317	734	12	**10**
18-78-2	n	18,720	22	22	22
bdd-21-133-	t	4,312	1,308	28	**22**
18-78-7	n	36,383	22	22	22
BDD	t	4,247	790	18	**16**
AVERAGE	n	34,691	17	17	17
renault-mod-5	t	327	-	883	**54**
	n	1,070	-	95	0
renault-mod-30	t	1,136	-	1,377	**529**
	n	1,468	-	870	882
modified Renault	t	138	-	378	**113**
AVERAGE	n	784	-	299	198

RmaxRPWC1 also managed to solve all instances, but it was typically outperformed by maxRPWC+, in some cases by very large margins (e.g. *Positive table-10*). These differences are due to the stronger pruning achieved by maxRPWC+ because of the check for lack of PW-support, as well as the cpu time gained by the fast check for PW-support and the efficient way of searching for PW-supports.

We also ran maxRPWC+ on two intensionally specified problems: *chessboard coloration* and *Schurr's lemma*. In this case we compared it against the generic GAC algorithm GAC2001/3.1 (Bessière et al. 2005). GAC2001/3.1 took 27 and 118 seconds on average to solve the instances of these classes respectively, while maxRPWC+ required 41 and 154 seconds. In both cases maxRPWC+ achieved very little, if any, extra pruning (in *Schurr's lemma* instances node visits where identical). Despite this, the cpu time overhead was not very significant. In contrast, maxRPWC+ clearly outperformed maxRPWC1 (947 and 298 secs), and was close to RmaxRPWC1 (52 and 155 secs).

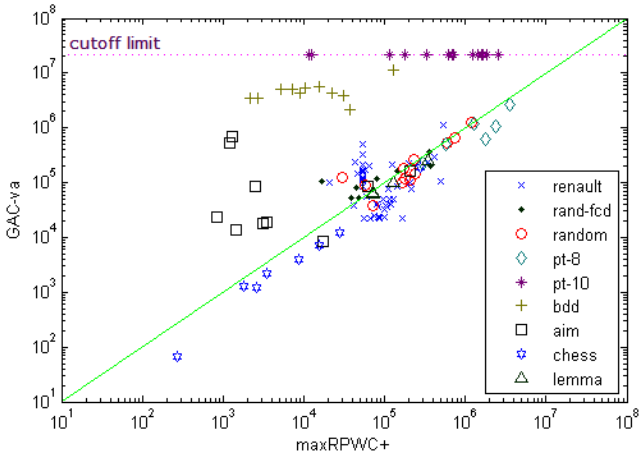

Figure 2: GAC-va vs. maxRPWC+.

Cpu times from all tested instances comparing maxRPWC+ to GAC-va (or GAC2001/3.1) are presented in Figure 2 in a logarithmic scale. Points above the diagonal correspond to instances that are solved faster by maxRPWC+. We can clearly see the benefits of our approach: Although most instances are gathered around the diagonal indicating closely matched performance, there are instances from various classes where GAC-va thrashes while maxRPWC+ does not. On the other hand, there are no instances where the opposite occurs. In a few words, maxRPWC+ is clearly more robust than a state-of-the-art GAC algorithm in classes of problems with interleaved table constraints, being orders of magnitude faster in some of these classes.

Related Work

GAC algorithms for table constraints have attracted considerable interest dating back to GAC-Schema (Bessière and Régin 1996). This generic method can be instantiated to either a method that searches the lists of allowed tuples for suppports, or to one that searches the valid tuples. The

GAC-va algorithm of (Lecoutre and Szymanek 2006) improves on GAC-Schema by interleaving the exploration of allowed and valid tuples using binary search. The interleaved exploration of allowed and valid tuples is also the main idea in (Lhomme and Régin 2005). However, in this case it is implemented through the use of an elaborate data structure (*Hologram*) introduced in (Lhomme 2004).

In (Gent et al. 2007) alternative data structures for table constraints were introduced with a Trie structure being the most efficient one. Also, authors in (Katsirelos and Walsh 2007) used a compact representation for allowed and disallowed tuples which can be constructed from a decision tree that represents the original tuples.

Simple Tabular Reduction (STR) (Ullmann 2007) and its variants (Lecoutre 2011) constitute an alternative, and efficient, approach to enforcing GAC based on the dynamic maintainance of the support tables. Finally, (Cheng and Yap 2010) uses multi-valued decision diagrams to store and process table constraints. Experimental results given in (Cheng and Yap 2010) show that the mdd approach is the fastest one. The algorithm of (Lecoutre and Szymanek 2006), on which we build, is very competitive with the Trie approach, outperforms the Hologram method and has the advantage of easier implementation and lack of complex data strucures over all other methods.

With respect to higher-order 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. (van Beek and Dechter 1995; Jégou 1993)). Quite recently, efficient ways to apply such consistencies were proposed (Karakashian et al. 2010), and new consistencies of this type were introduced (Lecoutre, Cardon, and Vion 2007). Domain filtering higher-order consistencies have also received attention recently (Bessiere, Stergiou, and Walsh 2008; Stergiou 2007).

Finally, a relevant work was presented in (Lhomme 2004) where a method to achieve GAC on a conjunction of two constraints using the *Hologram* data structure was proposed. However, the method was not evaluated experimentally.

Our paper binds together recent advances on GAC for table constraints and higher-order domain filtering consistencies contributing to both directions. Specifically, we offer an efficient method for strong filtering in cases of interleaved table constraints, and we make higher-order consistencies more practical by moving from generic to specialized algorithms. In the future we will explore the benefits of combining domain and relation filtering approaches to the handling of table constraints.

Conclusion

We presented maxRPWC+, an efficient specialized algorithm that achieves a higher order local consistency on table constraints. This algorithm builds on and extends existing algorithms for maxRWPC and GAC and can be easily adapted to operate on intensional constraints. In addition, it can be easily crafted into standard CP solvers. Experimental results demonstrated the practical usefulness of the proposed algorithm in the presence of interleaved table constraints, showing that maxRPWC+ is clearly more robust than a state-of-the-art GAC algorithm and considerably faster than generic maxRPWC algorithms.

References

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

Bessière, C., and Régin, J. 1996. Arc Consistency for General Constraint Networks: Preliminary Results. In *Proceedings of IJCAI'97*, 398–404.

Bessière, C.; Régin, J.; Yap, R.; and Zhang, Y. 2005. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence* 165(2):165–185.

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

Boussemart, F.; Heremy, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, 482–486.

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

Debruyne, R., and Bessière, C. 2001. Domain Filtering Consistencies. *JAIR* 14:205–230.

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

Jégou, P. 1993. On the Consistency of General Constraint Satisfaction Problems. In *Proceedings of AAAI'93*, 114–119.

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

Katsirelos, G., and Walsh, T. 2007. A compression algorithm for large arity extensional constraints. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, 379–393. Springer-Verlag.

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

Lecoutre, C.; Cardon, S.; and Vion, J. 2007. Conservative Dual Consistency. In *Proceedings of AAAI'07*, 237–242.

Lecoutre, C. 2011. Str2: optimized simple tabular reduction for table constraints. *Constraints* 16(4):341–371.

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

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

Stergiou, K. 2007. Strong inverse Consistencies for Non-Binary CSPs. In *Proceedings of ICTAI'07*, 215–222.

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

van Beek, P., and Dechter, R. 1995. On the Minimality and Global Consistency of Row-convex Constraint Networks. *JACM* 42(3):543–561.

Vion, J., and Debruyne, R. 2009. Light Algorithms for Maintaining Max-RPC During Search. In *Proceedings of SARA'09*.