Improving the Performance of maxRPC

Thanasis Balafoutis [1], Anastasia Paparrizou [2], Kostas Stergiou [2], and Toby Walsh [3]

[1]Department of Information and Communication Systems Engineering, University of the Aegean, Greece.
[2]Department of Informatics and Telecommunications Engineering, University of Western Macedonia, Greece.
[3]NICTA, University of New South Wales, Australia.

Abstract

Max Restricted Path Consistency (maxRPC) is a local consistency for binary constraints that can achieve considerably stronger pruning than arc consistency. However, existing maxRPC algorithms suffer from overheads and redundancies as they can repeatedly perform many constraint checks without triggering any value deletions. In this paper we propose techniques that can boost the performance of maxRPC algorithms. These include the combined use of two data structures to avoid many redundant constraint checks, and heuristics for the efficient ordering and execution of certain operations. Based on these, we propose two closely related maxRPC algorithms. The first one has optimal O(end^3) time complexity, displays good performance when used stand-alone, but is expensive to apply during search. The second one has O(en^2d^4) time complexity, but a restricted version with O(end^4) complexity can be very efficient when used during search. Both algorithms have O(ed) space complexity when used stand-alone. However, the first algorithm has O(end) space complexity when used during search, while the second retains the O(ed) complexity. Experimental results demonstrate that the resulting methods constantly outperform previous algorithms for maxRPC, often by large margins, and constitute a more than viable alternative to arc consistency.

1 Introduction

maxRPC is a strong domain filtering consistency for binary constraints introduced in 1997 by Debruyne and Bessiere [5]. maxRPC achieves a stronger level of local consistency than arc consistency (AC), and in [6] it was identified, along with

singleton AC (SAC), as a promising alternative to AC. Although SAC has received considerable attention since, maxRPC has been comparatively overlooked. The basic idea of maxRPC is to delete any value a of a variable x that has no arc consistency (AC) or path consistency (PC) support in a variable y. A value b is an AC support for a if the two values are compatible, and it is also a PC support for a if this pair of values is path consistent. A pair of values (a, b) is path consistent iff for every third variable there exists at least one value, called a PC witness, that is compatible with both a and b.

The first algorithm for maxRPC was proposed in [5], and two more algorithms have been proposed since then [7, 10]. The algorithms of [5] and [10] have been evaluated on random problems only, while the algorithm of [7] has not been experimentally evaluated at all. Despite achieving considerable pruning, existing maxRRC algorithms suffer from overhead and redundancies as they can repeatedly perform many constraint checks without triggering any value deletions. These constraint checks occur when a maxRPC algorithm searches for an AC support for a value and when, having located one, it checks if it is also a PC support by looking for PC witnesses in other variables. As a result, the use of maxRRC during search often slows down the search process considerably compared to AC, despite the savings in search tree size.

In this paper we propose techniques to improve the applicability of maxRPC by eliminating some of these redundancies while keeping a low space complexity. We also investigate approximations of maxRPC that only make slightly fewer value deletions in practice, while being significantly faster. We first demonstrate that we can avoid many redundant constraint checks and speed up the search for AC and PC supports through the careful and combined application of two data structures already used by maxRPC and AC algorithms [7, 10, 2, 8, 9]. Based on this, we propose a coarse-grained maxRPC algorithm called `maxRPC3` with optimal O(end^3) time complexity. This algorithm displays good performance when used stand-alone (e.g. for preprocessing), but is expensive to apply during search. We then propose another maxRPC algorithm, called `maxRPC3`^{rm}. This algorithm has O(en^2d^4) time complexity, but a restricted version with O(end^4) complexity can be very efficient when used during search through the use of *residues*. Both algorithms have O(ed) space complexity when used stand-alone. However, `maxRPC3` has O(end) space complexity when used during search, while `maxRPC3`^{rm} retains the O(ed) complexity.

Similar algorithmic improvements can be applied to *light maxRPC* (lmaxRPC), an approximation of maxRPC [10]. This achieves a lesser level of consistency compared to maxRPC but still stronger than AC, and is more cost-effective than maxRPC when used during search. Experiments confirm that lmaxRPC is indeed a considerably better option than maxRPC.

We also propose a number of heuristics that can be used to efficiently order the searches for PC supports and witnesses. Interestingly, some of the proposed heuristics not only reduce the number of constraint checks but also the number of visited nodes.

We make a detailed experimental evaluation of new and existing algorithms on various problem classes. This is the first wide experimental study of algorithms for maxRPC and its approximations on benchmark non-random problems. Results show that our methods constantly outperform existing algorithms, often by large margins. When applied during search our best method offers up to one order of magnitude reduction in constraint checks, while cpu times are improved up to four times compared to the best existing algorithm. In addition, these speed-ups enable a search algorithm that applies lmaxRPC to compete with or outperform MAC on many problems.

2 Background and Related Work

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, with maximum cardinality d, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c is a pair $(var(c), rel(c))$, where $var(c) = \{x_1, \ldots, x_m\}$ is an ordered subset of X, 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 $var(c)$. In the following, a binary constraint c with $var(c) = \{x_i, x_j\}$ will be denoted by c_{ij}, and $D(x_i)$ will denote the current domain of variable x_i. Each tuple $\tau \in rel(c)$ is an ordered list of values $(a_1, \ldots, a_m)$ such that $a_j \in D(x_j)$,$j = 1, \ldots, m$. A tuple $\tau \in rel(c_i)$ is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable.

The process which verifies whether a given tuple is allowed by a constraint c is called a *constraint check*. A binary CSP is a CSP where each constraint involves at most two variables. We assume that binary constraint checks are performed in constant time. In a binary CSP, a value $a_i \in D(x_i)$ is *arc consistent* (AC) iff for every constraint c_{ij} there exists a value $a_j \in D(x_j)$ s.t. the pair of values (a_i, a_j) satisfies c_{ij}. In this case a_j is called an *AC-support* of a_i. A variable is AC iff all its values are AC. A problem is AC iff there is no empty domain in D and all the variables in X are AC.

2.1 maxRPC

A value $a_i \in D(x_i)$ is *max restricted path consistent* (maxRPC) iff it is AC and for each constraint c_{ij} there exists a value $a_j \in D(x_j)$ that is an AC-support of a_i s.t. the pair of values (a_i, a_j) is *path consistent* (PC) [5]. A pair of values (a_i, a_j) is PC iff for any third variable x_k there exists a value $a_k \in D(x_k)$ s.t. a_k is an AC-support of both a_i and a_j. In this case a_j is a *PC-support* of a_i in x_j and a_k is a *PC-witness* for the pair (a_i, a_j) in x_k. A variable is maxRPC iff all its values are maxRPC. A problem is maxRPC iff there is no empty domain and all variables are maxRPC.

To our knowledge, three algorithms for achieving maxRPC have been proposed in the literature so far. The first one, called `maxRPC1`, is a fine-grained algorithm based on `AC6` and has optimal O(end^3) time complexity and O(end) space complexity [5]. The second algorithm, called `maxRPC2`, is a coarse-grained algorithm having O(end^3) time and O(ed) space complexity [7]. Finally, `maxRPC`^{rm} is a coarse-grained algorithm based on `AC3`^{rm} [10]. The time and space complexities of `maxRPC`^{rm} are O(en^2d^4) and O(end) but it has some advantages compared to the other two because of its lighter use of data structures. Among the three algorithms maxRPC2 seems to be the most promising for stand-alone use as it has a better time and space complexity than `maxRPC`^{rm} without requiring heavy data structures or complex implementation as `maxRPC1` does. On the other hand, `maxRPC`^{rm} can be better suited for use during search as it avoids the costly maintainance of data structures.

Central to `maxRPC2` is the $LastPC$ data structure, as we call it here. For each constraint c_{ij} and each value $a_i \in D(x_i)$, $LastPC_{x_i,a_i,x_j}$ gives the most recently discovered PC-support of a_i in $D(x_j)$. `maxRPC2` maintains this data structure incrementally. This means that the data structure is copied when moving forward during search (i.e. after a successfully propagated variable assignment) and restored when backtracking (after a failed variable assignment). This results in the following behavior: When looking for a PC-support for a_i in $D(x_j)$, `maxRPC2` first checks if $LastPC_{x_i,a_i,x_j}$ is valid. If it is not, it searches for a new PC-support starting from the value immediately after $LastPC_{x_i,a_i,x_j}$ in $D(x_j)$. In this way a good time complexity bound is achieved. On the other hand, `maxRPC`^{rm} uses a data structure similar to $LastPC$ to store *residues*, i.e. supports that have been discovered during execution and stored for future use, but does not maintain this structure incrementally[1]. When looking for a PC-support for a_i in $D(x_j)$, if the residue $LastPC_{x_i,a_i,x_j}$ is not valid then `maxRPC`^{rm} searches for a new PC-support from scratch in $D(x_j)$. This results in higher complexity, but crucially does not require costly maintainance of the $LastPC$ data structure during search.

[1] `maxRPC`^{rm} also uses residues in a different context.

A major overhead of both maxRPC2 and maxRPC^{rm} is the following. When searching for a PC-witness for a pair of values (a_i, a_j) in a third variable x_k, they always start the search from scratch, i.e. from the first available value in $D(x_k)$. As these searches can be repeated many times during search, there can be many redundant constraint checks. In contrast, maxRPC1 manages to avoid searching from scratch through the use of an additional data structure. This saves many constraint checks, albeit resulting in O(end) space complexity and requiring costly maintainance of this data structure during search. The algorithms we describe below largely eliminate these redundant constraint checks with lower space complexity, and in the case of maxRPC3^{rm} with only light use of data structures.

3 New Algorithms for maxRPC

We first recall the basic ideas of algorithms maxRPC2 and maxRPC^{rm} as described in [7] and [10]. Both algorithms use a propagation list L where variables whose domain is pruned are added. Once a variable x_j is removed from L all neighboring variables are revised to delete any values that are no longer maxRPC. For any value a_i of such a variable x_i there are two possible reasons for deletion. The first, which we call *PC-support loss* hereafter, is when the unique PC-support $a_j \in D(x_j)$ for a_i has been deleted. The second, which we call *PC-witness loss* hereafter, is when the unique PC-witness $a_j \in D(x_j)$ for the pair (a_i, a_k), where a_k is the unique PC-support for a_i on some variable x_k, has been deleted. In both cases value a_i is no longer maxRPC.

We now give a unified description of algorithms maxRPC3 and maxRPC3^{rm}. Both algorithms utilize data structures $LastPC$ and $LastAC$ which have the following functionalities: For each constraint c_{ij} and each value $a_i \in D(x_i)$, $LastPC_{x_i,a_i,x_j}$ and $LastAC_{x_i,a_i,x_j}$ give (point to) the most recently discovered PC and AC supports of a_i in $D(x_j)$ respectively. Initially, all $LastPC$ and $LastAC$ pointers are set to a special value NIL, considered to precede all values in any domain. Algorithm maxRPC3 updates the $LastPC$ and $LastAC$ structures incrementally like maxRPC2 and AC2001/3.1 respectively do. In contrast, algorithm maxRPC3^{rm} uses these structures as residues like maxRPC^{rm} and AC^{rm} do.

The pseudocode for the unified description of maxRPC3 and maxRPC3^{rm} is given in Algorithm 1 and Functions 2, 3, 4. We assume the existence of a global Boolean variable RM which determines whether the algorithm presented is instantiated to maxRPC3 or to maxRPC3^{rm}. If RM is true, the algorithm used is maxRPC3^{rm}. Otherwise, the algorithm is maxRPC3.

Being coarse-grained, Algorithm 1 uses a propagation list L where variables that have their domain filtered are inserted. If the algorithm is used for preprocess-

ing then, during an initialization phase, for each value a_i of each variable x_i we check if a_i is maxRPC. If it is not then it is deleted from $D(x_i)$ and x_i is added to L. The initialization function is not shown in detail due to limited space. If the algorithm is used during search then L is initialized with the currently assigned variable (line 3).

In the main part of Algorithm 1, when a variable x_j is removed from L, each variable x_i constrained with x_j must be made maxRPC. For each value $a_i \in D(x_i)$ Algorithm 1, like maxRPC2 and maxRPC^{rm}, establishes if a_i is maxRPC by checking for PC-support loss and PC-witness loss at lines 8 and 12.

Algorithm 1 maxRPC3/maxRPC3^{rm}

```
 1: if ¬ RM then
 2:     if ¬initialization(L, LastPC, LastAC) then return FAILURE;
 3: else L = {currently assigned variable};
 4: while L ≠ ∅ do
 5:     L=L−{x_j};
 6:     for each x_i ∈ X s.t. c_ij ∈ C do
 7:         for each a_i ∈ D(x_i) do
 8:             if ¬searchPCsup(a_i, x_j) then
 9:                 delete a_i;
10:                 L=L ∪ {x_i};
11:             else
12:                 if ¬checkPCwit(a_i, x_j) then
13:                     delete a_i;
14:                     L=L ∪ {x_i};
15:         if D(x_i) is empty then return FAILURE;
16: return SUCCESS;
```

First, function *searchPCsup* is called to check if a PC-support for a_i exists in $D(x_j)$. If value $LastPC_{x_i,a_i,x_j}$ is still in $D(x_j)$, then *searchPCsup* returns TRUE (lines 1-2). If $LastPC_{x_i,a_i,x_j}$ is not valid, we search for a new PC-support. If maxRPC3 is used, we can take advantage of the $LastPC$ and $LastAC$ pointers to avoid starting this search from scratch. Specifically, we know that no PC-support can exist before $LastPC_{x_i,a_i,x_j}$, and also none can exist before $LastAC_{x_i,a_i,x_j}$, since all values before $LastAC_{x_i,a_i,x_j}$ are not AC-supports of a_i. Lines 5-6 in *searchPCsup* take advantage of these to locate the appropriate starting value b_j. Note that maxRPC2 always starts the search for a PC-support from the value after $LastPC_{x_i,a_i,x_j}$. If the algorithm is called during search, in which case we use maxRPC3^{rm} then the search for a new PC-support starts from scratch (line 8), just like maxRPC^{rm} does.

For every value $a_j \in D(x_j)$, starting with b_j, we first check if is an AC-support of a_i (line 10). This is done using function *isConsistent* which simple checks if two values are compatible. If it is, and the algorithm is maxRPC3, then we can update $LastAC_{x_i,a_i,x_j}$ under a certain condition (lines 12-13). Specifically,

if $LastAC_{x_i,a_i,x_j}$ was deleted from $D(x_j)$, then we can set $LastAC_{x_i,a_i,x_j}$ to a_j in case $LastAC_{x_i,a_i,x_j} > LastPC_{x_i,a_i,x_j}$. If $LastAC_{x_i,a_i,x_j} \leq LastPC_{x_i,a_i,x_j}$ then we cannot do this as there may be AC-supports for a_i between $LastAC_{x_i,a_i,x_j}$ and $LastPC_{x_i,a_i,x_j}$ in the lexicographical ordering. We then move on to verify the path consistency of (a_i, a_j) through function *searchPCwit*.

If no PC-support for a_i is found in $D(x_j)$, *searchPCsup* will return FALSE, a_i will be deleted and x_i will be added to L. Otherwise, $LastPC_{x_i,a_i,x_j}$ is set to the discovered PC-support a_j (line 15). If maxRPC3^{rm} is used then we update the residue $LastAC_{x_i,a_i,x_j}$ since the discovered PC-support is also an AC-support. In addition, to exploit the multidirectionality of residues, maxRPC3^{rm} sets $LastPC_{x_j,a_j,x_i}$ to a_i, as in [10].

Function 2 *searchPCsup*(a_i, x_j):**boolean**

```
1: if LastPC_{x_i,a_i,x_j} ∈ D(x_j) then
2:     return true;
3: else
4:     if ¬ RM then
5:         if LastAC_{x_i,a_i,x_j} ∈ D(x_j) then b_j = max(LastPC_{x_i,a_i,x_j}+1,LastAC_{x_i,a_i,x_j});
6:         else b_j = max(LastPC_{x_i,a_i,x_j}+1,LastAC_{x_i,a_i,x_j}+1);
7:     else
8:         b_j = first value in D(x_j);
9:     for each a_j ∈ D(x_j), a_j ≥ b_j do
10:        if isConsistent(a_i, a_j) then
11:            if ¬RM then
12:                if LastAC_{x_i,a_i,x_j} ∉ D(x_j) AND LastAC_{x_i,a_i,x_j} > LastPC_{x_i,a_i,x_j} then
13:                    LastAC_{x_i,a_i,x_j} = a_j;
14:            if searchPCwit(a_i, a_j) then
15:                LastPC_{x_i,a_i,x_j} = a_j;
16:                if RM then LastAC_{x_i,a_i,x_j} = a_j; LastPC_{x_j,a_j,x_i} = a_i;
17:                return true;
18: return false;
```

Function *searchPCwit* checks if a pair of values (a_i,a_j) is PC by doing the following for each variable x_k constrained with x_i and x_j[2]. First, it checks if either $LastAC_{x_i,a_i,x_k}$ is valid and consistent with a_j or $LastAC_{x_j,a_j,x_k}$ is valid and consistent with a_i (line 3). If one of these conditions holds then we have found a PC-witness for (a_i,a_j) without searching in $D(x_k)$ and we move on to the next variable constrained with x_i and x_j. Note that neither maxRPC2 nor maxRPC^{rm} can do this as they do not have the $LastAC$ structure. Experimental results in Section 5 demonstrate that these simple conditions can eliminate a very large number of redundant constraint checks.

If none of the conditions holds then we have to search in $D(x_k)$ for a PC-witness. If the algorithm is maxRPC3 then we can exploit the $LastAC$ structure

[2]Since AC is enforced by the maxRPC algorithm, we only need to consider variables that form a 3-clique with x_i and x_j.

to start this search from $b_k = max\{LastAC_{x_i,a_i,x_k}, LastAC_{x_j,a_j,x_k}\}$ (line 6). But before doing this, we call function *seekACsupport* (not shown for space reasons), first with (x_i, a_i, x_k) and then with (x_j, a_j, x_k) as parameters, to find the lexicographically smallest AC-supports for a_i and a_j in $D(x_k)$ (line 5). If such supports are found, $LastAC_{x_i,a_i,x_k}$ and $LastAC_{x_j,a_j,x_k}$ are updated accordingly. In case no AC-support is found for either a_i or a_j then *seekACsupport* returns FALSE, and subsequently searchPCwit() will also return FALSE.

Function 3 *searchPCwit*(a_i, a_j):**boolean**

```
1: for each x_k ∈ V s.t. c_ik ∈ C and c_jk ∈ C do
2:     maxRPCsupport=FALSE;
3:     if (LastAC_{x_i,a_i,x_k} ∈ D(x_k) AND isConsistent(LastAC_{x_i,a_i,x_k}, a_j)) OR (LastAC_{x_j,a_j,x_k} ∈
       D(x_k) AND isConsistent(LastAC_{x_j,a_j,x_k}, a_i)) then continue;
4:     if ¬ RM then
5:         if ¬seekACsupport(x_i, a_i, x_k) OR ¬seekACsupport(x_j, a_j, x_k) then return false;
6:         b_k = max(LastAC_{x_i,a_i,x_k}, LastAC_{x_j,a_j,x_k});
7:     else b_k = first value in D(x_k);
8:     for each a_k ∈ D(x_k), a_k ≥ b_k do
9:         if isConsistent(a_i, a_k) AND isConsistent(a_j, a_k) then
10:            if RM then LastAC_{x_i,a_i,x_k} = LastAC_{x_j,a_j,x_k} = a_k;
11:            maxRPCsupport=TRUE; break;
12:    if ¬maxRPCsupport then return false;
13: return true;
```

If the algorithm used is maxRPC3^{rm} then we start search for a PC-witness from scratch (line 7), as maxRPC2 and maxRPC^{rm} always do. If a PC-witness a_k is found (line 9) and we are using maxRPC3^{rm} then both residues $LastAC_{x_i,a_i,x_k}$ and $LastAC_{x_j,a_j,x_k}$ are set to a_k as they are the most recently discovered AC-supports. If no PC-witness is found then we have determined that the pair (a_i,a_j) is not PC and as a result FALSE will be returned and *searchPCsup* will move to check if the next available value in $D(x_j)$ is a PC-support for a_i.

If value a_i is not removed by *searchPCsup* in Algorithm 1, *checkPCwit* is called to check for PC-witness loss. This is done by iterating over the variables that are constrained with both x_i and x_j. For each such variable x_k, we first check if $a_k = LastPC_{x_i,a_i,x_k}$ is still in $D(x_k)$ (line 3). If so then we check if there still is a PC-witness in $D(x_j)$. This is done by first checking if either $LastAC_{x_i,a_i,x_j}$ is valid and consistent with a_k or $LastAC_{x_k,a_k,x_j}$ is valid and consistent with a_i (line 4). If neither of these conditions holds then we search for a PC-witness starting from $b_j = max\{LastAC_{x_i,a_i,x_j}, LastAC_{x_k,a_k,x_j}\}$ in case of maxRPC3 (line 9), after checking the existence of AC-supports for a_i and a_k in $D(x_j)$, by calling *seekACsupport* (line 8). If there is no AC-support in $D(x_j)$ for either a_i or a_k we set the auxiliary Boolean variable *findPCsupport* to TRUE to avoid searching for a PC-witness.

If maxRPC3^{rm} is used, we start searching for a PC-witness from scratch (line

11). Note that maxRPC2 does not do the check of line 4 and always starts the search for a PC-witness from the first value in $D(x_j)$. In contrast, maxRPC^{rm} avoids some redundant checks through the use of special residues, albeit resulting in O(end) space complexity. When using maxRPC3^{rm}, for each value $a_j \in D(x_j)$ we check if it is compatible with a_i and a_k and move the $LastAC$ pointers accordingly (lines 14-15), exploiting the multidirectionality of residues,

Function 4 *checkPCwit*(a_i, x_j):**boolean**

```
1: for each x_k ∈ V s.t. c_ik ∈ C and c_kj ∈ C do
2:     witness=FALSE; findPCsupport=FALSE;
3:     if a_k = LastPC_{x_i,a_i,x_k} ∈ D(x_k) then
4:         if (LastAC_{x_i,a_i,x_j} ∈ D(x_j) AND isConsistent(LastAC_{x_i,a_i,x_j}, a_k)) OR (LastAC_{x_k,a_k,x_j} ∈
           D(x_j) AND isConsistent(LastAC_{x_k,a_k,x_j}, a_i)) then
5:             witness=TRUE;
6:         else
7:             if ¬ RM then
8:                 if seekACsupport(x_i, a_i, x_j) AND seekACsupport(x_k, a_k, x_j) then
9:                     b_j = max(LastAC_{x_i,a_i,x_j}, LastAC_{x_k,a_k,x_j});
10:                else findPCsupport=TRUE;
11:            else b_j = first value in D(x_j);
12:            if ¬findPCsupport then
13:                for each a_j ∈ D(x_j), a_j ≥ b_j do
14:                    if isConsistent(a_i, a_j) AND isConsistent(a_k, a_j) then
15:                        if RM then LastAC_{x_i,a_i,x_j} = LastAC_{x_k,a_k,x_j} = a_j;
16:                        witness=TRUE; break;
17:    if ¬witness AND exists a_k > LastPC_{x_i,a_i,x_k} ∈ D(x_k) then
18:        if ¬ RM then
19:            if LastAC_{x_i,a_i,x_k} ∈ D(x_k) then b_k = max(LastPC_{x_i,a_i,x_k}+1,LastAC_{x_i,a_i,x_k});
20:            else b_k = max(LastPC_{x_i,a_i,x_k}+1,LastAC_{x_i,a_i,x_k}+1
21:        else
22:            b_k = first value in D(x_k);
23:        for each a_k ∈ D(x_k), a_k ≥ b_k do
24:            if isConsistent(a_i, a_k) then
25:                if ¬ RM then
26:                    if LastAC_{x_i,a_i,x_k} ∉ D(x_k) AND LastAC_{x_i,a_i,x_k} > LastPC_{x_i,a_i,x_k} then
27:                        LastAC_{x_i,a_i,x_k} = a_k;
28:                if searchPCwit(a_i, a_k) then
29:                    LastPC_{x_i,a_i,x_k} = a_k;
30:                    if RM then LastAC_{x_i,a_i,x_k} = a_k; LastPC_{x_k,a_k,x_i} = a_i;
31:                    witness=TRUE; break;
32:    if ¬witness then return false;
33: return true;
```

If $LastPC_{x_i,a_i,x_k}$ has been removed or a_i has no PC-witness in $D(x_j)$, we search for a new PC-support for a_i in $D(x_k)$. As in function *searchPCsup*, when maxRPC3 is used this search starts at an appropriate value calculated taking advantage of $LastPC_{x_i,a_i,x_k}$ and $LastAC_{x_i,a_i,x_k}$ (lines 18-20). When maxRPC3^{rm} is used we start from scratch. If an AC-support for a_i is found (line 24), we check if it is also a PC-support by calling function *searchPCwit* (line 28). If maxRPC3 is used then $LastAC_{x_i,a_i,x_k}$ is updated when necessary (lines 26-27). If a PC-support

is found, $LastPC_{x_i,a_i,x_k}$ is set accordingly (line 29). If maxRPC3^{rm} is used then the residue $LastAC_{x_i,a_i,x_k}$ is also updated, as is $LastPC_{x_k,a_k,x_i}$ (bidirectionality). If the search for a PC-support fails then FALSE will be returned, a_i will be deleted, and x_i will be added to L.

3.1 Light maxRPC

Light maxRPC (lmaxRPC) is an approximation of maxRPC that only propagates the loss of AC-supports and not the loss of PC-witnesses [10]. This ensures that the obtained algorithm enforces a consistency property that is at least as strong as AC.

lmaxRPC is a procedurally defined local consistency, meaning that its description is tied to a specific maxRPC algorithm. Light versions of algorithms maxRPC3 and maxRPC3^{rm}, simply noted lmaxRPC3 and lmaxRPC3^{rm} respectively, can be obtained by omitting the call to the *checkPCwit* function (lines 11-14 of Algorithm 1). In a similar way, we can obtain light versions of algorithms maxRPC2 and maxRPC^{rm}.

As already noted in [10], the light versions of different maxRPC algorithms may not be equivalent in terms of the pruning they achieve. To give an example, a brute force algorithm for lmaxRPC that does not use any data structures can achieve more pruning than algorithms lmaxRPC2, lmaxRPC3, and lmaxRPC^{rm}, albeit being much slower in practice. Consider that any of these three algorithms will return TRUE in case $LastPC_{x_i,a_i,x_j}$ is valid. However, although $LastPC_{x_i,a_i,x_j}$ is valid, it may no longer be a PC-support because the PC-witness in some third variable may have been deleted, and it may be the last one. In a case where $LastPC_{x_i,a_i,x_j}$ was the last PC-support in x_j for value a_i, the three advanced algorithms will not delete a_i while the brute force one will. This is because it will exhaustively check all values of x_j for PC-support, concluding that there is none.

The worst-case time and space complexities of algorithm lmaxRPC2 are the same as maxRPC2. Algorithm lmaxRPC^{rm} has O(n^3d^4) time and O(ed) space complexities, which are lower than those of maxRPC^{rm}. Experiments with random problems using algorithms lmaxRPC^{rm} and maxRPC^{rm} showed that the pruning power of lmaxRPC is only slightly weaker than that of maxRPC [10]. At the same time, it can offer significant gains in run times when used during search. These results were also verified by us through a series of experiments on various problem classes.

3.2 Correctness and Complexities

We now prove the correctness of algorithms maxRPC3 and maxRPC3^{rm} and analyze their worst-case time and space complexities.

Proposition 1 Algorithm maxRPC3 is sound and complete.

Proof: ***Soundness.*** To prove the soundness of maxRPC3 we must prove that any value that is deleted by maxRPC3 is not maxRPC. Let $a_i \in D(x_i)$ be a value that is deleted by maxRPC3. It is either removed from $D(x_i)$ during the initialization phase (line 15) or in line 8 of Algorithm 1, after *searchPCsup* has returned *false*, or in line 12, after *searchPCsup* has returned *true* and *checkPCwit* has returned *false*.

In the first case, since function *initilization* checks all values in a brute-force manner, it is clear that any deleted value a_i either has no AC-support or none of its AC-supports is a PC-support in some variable x_j. The non-existence of a PC-support is determined using function *searchPCwit* whose correctness is discussed below.

In the second case, since *searchPCsup* returns *false*, $LastPC_{x_i,a_i,x_j}$ is not valid so a new PC-support in $D(x_j)$ is seeked (lines 9-17). This search starts with the value at max($LastPC_{x_i,a_i,x_j}$+1, $LastAC_{x_i,a_i,x_j}$) or at max($LastPC_{x_i,a_i,x_j}$ +1, $LastAC_{x_i,a_i,x_j}$ +1), depending on whether $LastAC_{x_i,a_i,x_j}$ is valid or not. This is correct since any value before $LastPC_{x_i,a_i,x_j}$+1 and any value before $LastAC_{x_i,a_i,x_j}$ is definitely not an AC-support for a_i (similarly for the other case). *searchPCsup* will return false either because no AC-support for a_i can be found in $D(x_j)$ (line 10), or because for any AC-support found, *searchPCwit* returned false (line 13). In the former case there is no PC-support for a_i in $D(x_j)$ since there is no AC-support. In the latter case, for any AC-support a_j found there must be some third variable x_k for which no PC-witness for the pair (a_i, a_j) exists. For each third variable x_k *searchPCwit* correctly identifies a PC-witness if one of the conditions in line 3 holds. In none holds then *searchPCwit* searches for a PC-witness starting from max($LastAC_{x_i,a_i,x_k}$, $LastAC_{x_j,a_j,x_k}$) (line 6). This is correct since $LastAC_{x_i,a_i,x_k}$ and $LastAC_{x_j,a_j,x_k}$ are updated with the lexicographically smallest support of a_i (resp. a_j) in $D(x_k)$ by calling function *seekACsup*, meaning that any value smaller than max($LastAC_{x_i,a_i,x_k}$, $LastAC_{x_j,a_j,x_k}$) is incompatible with either a_i or a_j. Therefore, if *searchPCwit* returns *false* then there is no PC-witness for some third variable x_k. Hence, if *searchPCsup* returns *false*, it means no PC-support for a_i can be found in $D(x_j)$ and it is thus correctly deleted.

Now assume that the call to *searchPCsup* returned *true* and a_i was removed after *checkPCwit* returned *false*. This means that for some variable x_k, constrained with both x_i and x_j, both the first part (lines 3-11) and the second part (lines 13-24) of *checkPCwit* failed to set the Boolean *witness* to *true*. Regarding the first part, the failure means that the pair of values (a_i, a_k), where a_k is the last PC-support of a_i in $D(x_k)$ found, has no PC-witness in $D(x_j)$. In more detail, the search for a PC-witness correctly starts from max($LastAC_{x_i,a_i,x_j}$, $LastAC_{x_j,a_j,x_j}$) in line 9, after both $LastAC$ pointers have been updated by *seekACsup*. The condition

in line 4 is similar to the corresponding condition in *searchPCwit* and thus, if it is true, the search for PC-witness is correctly overriden. Regarding the second part, the failure means that no alternative PC-support for a_i in $D(x_k)$ was found. In more detail, the search for a PC-support starts from max($LastPC_{x_i,a_i,x_k}$+1, $LastAC_{x_i,a_i,x_k}$) or max($LastPC_{x_i,a_i,x_k}$+1, $LastAC_{x_i,a_i,x_k}$+1), depending on the existence of $LastAC_{x_i,a_i,x_k}$. This is correct since no ealier value can be a PC-support. If there is no consistent (a_i, a_k) pair or *searchPCwit* returns *false* for all consistent pairs found, then a_i has no PC-support in $D(x_k)$ and is thus correctly deleted.

Completeness. To prove the completeness of `maxRPC3` we need to show that if a value is not maxRPC then the algorithm will delete it. The initialization function checks all values of all variables one by one in a brute-force manner and removes any value that is not maxRPC. Values that are maxRPC have their $LastPC$ pointers set to the discovered PC-supports. Thereafter, the effects of such removals are propagated by calling Algorithm 1 and as a result new value deletions may occur. Now consider a value $a_i \in D(x_i)$ that was not removed by the initialization function but after propagation is no longer maxRPC. This is either because of PC-support or PC-witness loss.

In the first case assume that x_j is the variable in which a_i no longer has a PC-support. Since the previously found PC-support of a_i has been deleted, x_j must have been added to Q at some point. When x_j is removed from Q all neighboring variables, including x_i will be checked. Function *searchPCsup* will find that $LastPC_{x_i,a_i,x_j}$ is no longer valid and will search for a new PC-support concluding that there is none. Therefore, it will return *false* and a_i will be deleted.

In the second case assume that the pair of values (a_i,a_j), where a_j is the last PC-support of a_i in $D(x_j)$, has lost its last PC-witness a_k in variable x_k. If $LastPC_{x_i,a_i,x_j}$ is not valid, which means that x_j was added to Q, then we have the same case as above. Therefore, after x_j is removed from Q, *searchPCsup* will find out that there is no PC-support for a_i in $D(x_j)$ and will delete it. If $LastPC_{x_i,a_i,x_j}$ is valid then *searchPCsup* will return *true* (line 2). Since a_k was deleted, x_k was added to Q at some point. When x_k is removed from Q all neighboring variables, including x_i will be checked. If a_i has no longer a PC-support in $D(x_k)$, this will be detected by *searchPCsup* and a_i will be deleted. Otherwise, function *checkPCwit* will be called. The for loop in line 1 will go through every variable constrained with both x_i and x_k, including x_j. Since $LastPC_{x_i,a_i,x_j}$ is valid, a new PC-witness for (a_i,a_j) in $D(x_k)$ will be seeked (lines 3-11). Since a_k was the last PC-witness, none will be found and as a result a new PC-support for a_i in $D(x_j)$ will be seeked (lines 13-24). Since a_j was the last PC-support for a_i in $D(x_j)$, none will be found, *checkPCwit* will return *false*, and a_i will be deleted.

Proposition 2 Algorithm maxRPC3^{rm} is sound and complete.

Proof: The proof is very similar to the corresponding proof for maxRPC3. As explained, the main difference between the two algorithms concerns the use of the $LastAC$ and $LastPC$ structures. As maxRPC3^{rm} does not maintain these structures incrementally, the searches for PC-supports in *searchPCsup* and *checkPCwit* and the searches for PC-witnesses in *searchPCwit* and *checkPCwit* start from scratch. Clearly, this has no effect on the soundness or completeness of the algorithm since it guarantees that all potential PC-supports and PC-witnesses are checked. Furthermore, the conditions for avoiding redundant searches using residues are the same as in maxRPC3. Finally, another difference between the two algorithms is the exploitation of bidirectionality by maxRPC3^{rm}. By the definition of path and arc consistency, bidirectionality holds. That is, when a PC-support (AC-support) $a_j \in D(x_j)$ is located for a value $a_i \in D(x_i)$ then a_i is a PC-support (AC-support) for a_j. Since the property of bidirectionality is exploited only to update residues, it does not affect the correctness of the algorithm.

We now discuss the complexities of algorithms maxRPC3 and maxRPC3^{rm} and their light versions. To directly compare with existing algorithms for (l)maxRPC, the time complexities give the asymptotic number of constraint checks[3]. Folllowing [9], the *node* time (resp. space) complexity of a (l)maxRPC algorithm is the worst-case time (resp. space) complexity of invoking the algorithm after a variable assignment. The corresponding *branch* complexities of an (l)maxRPC algorithm are the worst-case complexities of any incremental sequence of $k \leq n$ invocations of the algorithm. That is, the complexities of incrementally running the algorithm down a branch of the search tree until a fail occurs.

Proposition 3 The node and branch time complexity of (l)maxRPC3 is O(end^3).

Proof: The complexity is determined by the total number of calls to function *isConsistent* in *searchPCsup*, *checkPCwit*, and mainly *searchPCwit* where most checks are executed.

Each variable can be inserted and extracted from L every time a value is deleted from its domain, giving O(d) times in the worst case. Each time a variable x_j is extracted from L, *searchPCsup* will look for a PC-support in $D(x_j)$ for all values $a_i \in D(x_i)$, s.t. $c_{i,j} \in C$. For each variable x_i, O(d) values are checked. Checking if a value $a_j \in D(x_j)$ is a PC-support involves first checking in O(1) if it is an AC-support (line 9 in *searchPCsup*) and then calling *searchPCwit*. The cost of

[3]However, constraint checks do not always reflect run times as other operations may have an equal or even greater effect.

searchPCwit is O($n + nd$) since there are O(n) variables constrained with both x_i and x_j and, after making the checks in line 3, their domains must be searched for a PC-witness, each time from scratch with cost O(nd). Through the use of $LastPC$ no value of x_j will be checked more than once over all the O(d) times x_j is extracted from L, meaning that for any value $a_i \in D(x_i)$ and any variable x_j, the overall cost of *searchPCwit* will be O($dn + nd^2$) = O(nd^2). Hence, *searchPCsup* will cost O(nd^2) for one value of x_i, giving O(nd^3) for d values. Since, in the worst case, this process will be repeated for every pair of variables x_i and x_j that are constrained, the total cost of *searchPCsup* will be O(end^3). This is the node complexity of `lmaxRPC3`.

In *checkPCwit* the algorithms iterate over the variables in a triangle with x_j and x_i. In the worst case, for each such variable x_k, $D(x_j)$ will be searched from scratch for a PC-witness of a_i and its current PC-support in x_k. As x_j can be extracted from L O(d) times and each search from scratch costs O(d), the total cost of checking for a PC-witness in $D(x_j)$, including the checks of line 4 in *checkPCwit*, will be O($d + d^2$). For d values of x_i this will be O(d^3). As this process will be repeated for all triangles of variables, whose number is bounded by en, its total cost will be O(end^3). If no PC-witness is found then a new PC-support for a_i in $D(x_k)$ is seeked through *searchPCwit*. This costs O(nd^2) as explained above but it is amortized with the cost incurred by the calls to *searchPCwit* from *searchPCsup*. Therefore, the cost of *checkPCwit* is O(end^3). This is also the node complexity of `maxRPC3`.

The branch complexity of `(l)maxRPC3` is also O(end^3). This is because the use of $LastPC$ ensures that for any constraint $c_{i,j}$ and a value $a_i \in D(x_i)$, each value of x_j will be checked at most once for PC-support while going down the branch. Therefore, the cost of $searchPCwit$ is amortized.

Proposition 4 The node and branch time complexities of `lmaxRPC3`^{rm} and `maxRPC3`^{rm} are O(end^4) and O(en^2d^4) respectively.

Proof: The proof is similar to that of Proposition 3. The main difference with `lmaxRPC3` is that since $lastPC$ is not updated incrementally, each time we seek a PC-support for a value $a_i \in D(x_i)$ in x_j, $D(x_j)$ will be searched from scratch in the worst case. This incurs an extra O(d) cost to *searchPCsup* and *searchPCwit*. Hence, the node complexity of `lmaxRPC3`^{rm} is O(end^4). Also, the total cost of *searchPCwit* in one node cannot be amortized. This means that the cost of *searchPCwit* within *checkPCwit* is O(nd^2). Hence, the node complexity of `maxRPC3`^{rm} is O(en^2d^4). The branch complexities are the same because the calls to *searchPCwit* are amortized.

The space complexities of the algorithms are determined by the space required for data structures $LastPC$ and $LastAC$. Since both require O(ed) space, this is the node space complexity of `(l)maxRPC3` and `(l)maxRPC3`^{rm}. `(l)maxRPC3` has O(end) branch space complexity because of the extra space required for the incremental update and restoration of the data structures. As `(l)maxRPC3`^{rm} avoid this, its branch space complexity is O(ed).

4 Heuristics for maxRPC Algorithms

Numerous heuristics for ordering constraint or variable revisions have been proposed and used within AC algorithms [11, 3, 1]. Heuristics such as the ones used by AC algorithms can be also used within a maxRPC algorithm to efficiently select the next variable to be removed from the propagation list (line 5 of Algorithm 1). In addition to this, maxRPC and lmaxRPC algorithms can benefit from the use of heuristics elsewhere in their execution. Once a variable x_j has been removed from the propagation list, heuristics can be applied as follows in either a maxRPC or a lmaxRPC algorithm (we use algorithm `(l)maxRPC3` for illustration):

1. After a variable x_j is removed from L all neighboring variables x_i are revised. lmaxRPC (resp. maxRPC) will detect a failure if the condition of PC-support loss (resp. either PC-support or PC-witness loss) occurs for all values of x_i. In such situations, the sooner x_i is considered and the failure is detected, the more constraint checks will be saved. Hence, the order in which the neighboring variables of x_j are considered can be determined using a fail-first type of heuristic.

2. Once an AC-support $a_j \in D(x_j)$ has been found for a value $a_i \in D(x_i)$, *searchPCsup* tries to establish if it is a PC-support. If there is no PC-witness for the pair (a_i, a_j) in some variable x_k then a_j is not a PC-support. Therefore, we can again use fail-first heuristics to determine the order in which the variables forming a triangle with x_i and x_j are considered.

The above cases apply to both lmaxRPC and maxRPC algorithms. In addition, a maxRPC algorithm can employ heuristics as follows:

3. For each value $a_i \in D(x_i)$ and each variable x_k constrained with both x_i and x_j, Function 4 checks if the pair (a_i, a_k) still has a PC-witness in $D(x_j)$. If there is no PC-witness or $LastPC_{x_i,a_i,x_k}$ is not valid then a new PC-support in x_k is seeked. If none is found then a_i will be deleted. Again heuristics can be used to determine the order in which the variables constrained with x_i and x_j are considered.

4. In Function 4 if $LastPC_{x_i,a_i,x_k}$ is not valid then a new PC-support for a_i in $D(x_k)$ is seeked. The order in which variables constrained with both x_i and x_k are considered can be determined heuristically as in Case 2 above.

As explained, the purpose of such ordering heuristic will be to "fail-first". That is, to quickly discover potential failures (Case 1 above), refute values that are not PC-supports (Cases 2 and 4) and delete values that have no PC-support (Case 3). Such heuristics can be applied in any coarse-grained maxRPC algorithm to decide the order in which variables are considered in Cases 1-4. Examples are the following:

dom Consider the variables in ascending domain size. This heuristic can be applied in any of the four cases.

del_ratio Consider the variables in ascending ratio of the number of remaining values to the initial domain size. This heuristic can be applied in any of the four cases.

wdeg In Case 1 consider the variables x_i in descending weight for the constraint c_{ij}. In Case 2 consider the variables x_k in descending average weight for the constraints c_{ik} and c_{jk}. Similarly for Cases 3 and 4.

dom/wdeg Consider the variables in ascending value of dom/wdeg. This heuristic can be applied in any of the four cases.

Experiments demonstrated that applying heuristics in Cases 1 and 3 are particularly effective, while doing so in Cases 2 and 4 saves constraint checks but only marginally reduces cpu times. All of the heuristics mentioned above for Cases 1 and 3 offer cpu gains, with dom/wdeg being the most efficient. Although the primal purpose of the heuristics is to save constraint checks, it is interesting to note that some of the heuristics can also divert search to different areas of the search space when a variable ordering heuristic like dom/wdeg is used, resulting in fewer node visits. For example, two different orderings of the variables in Case 1 may result in different constraints causing a failure. As dom/wdeg increases the weight of a constraint each time it causes a failure and uses the weights to select the next variable, this may later result in different branching choices. This is explained for the case of AC in [1].

5 Experiments

We have experimented with several classes of structured and random binary CSPs taken from C.Lecoutre's XCSP repository. Excluding instances that were very hard

for all algorithms, our evaluation was done on 200 instances in total from various problem classes. More details about these instances can be found in C.Lecoutre's homepage. All algorithms used the dom/wdeg heuristic for variable ordering [4] and lexicographic value ordering. In case of a failure (domain wipe-out) the weight of constraint c_{ij} is updated (right before returning in line 15 of Algorithm 1). The suffix '+H' after any algorithm's name means that we have applied the dom/wdeg heuristic for ordering the propagation list [1], and the same heuristic for *Case 1* described in Section 4. In absense of the suffix, the propagation list was implemented as a FIFO queue and no heuristic from Section 4 was used.

Table 1: Average stand-alone performance in all 200 instances grouped by problem class. Cpu times (t) in secs and constraint checks (cc) are given.

Problem class		maxRPC2	maxRPC3	lmaxRPC2	lmaxRPC3	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC3+H
RLFAP	t	6.786	2.329	4.838	**2.043**	4.615	2.058	2.148
(scen,graph)	cc	31M	9M	21M	8M	21M	9M	8M
Random	t	0.092	0.053	0.079	0.054	0.078	**0.052**	0.056
(modelB,forced)	cc	0.43M	0.18M	0.43M	0.18M	0.43M	0.18M	0.18M
Geometric	t	0.120	0.71	0.119	0.085	0.120	0.086	**0.078**
	cc	0.74M	0.35M	0.74M	0.35M	0.74M	0.35M	0.35M
Quasigroup	t	0.293	0.188	0.234	0.166	0.224	**0.161**	0.184
(qcp,qwh,bqwh)	cc	1.62M	0.59M	1.28M	0.54M	1.26M	0.54M	0.54M
QueensKnights,	t	87.839	47.091	91.777	45.130	87.304	43.736	**43.121**
Queens,QueenAttack	cc	489M	188M	487M	188M	487M	188M	188M
driver,blackHole	t	0.700	0.326	0.630	**0.295**	0.638	0.303	0.299
haystacks,job-shop	cc	4.57M	1.07M	4.15M	1.00M	4.15M	1.00M	1.00M

Table 1 compares the performance of stand-alone algorithms used for preprocessing. We give average results for all the instances, grouped into specific problem classes. We include results from the two optimal coarse-grained maxRPC algorithms, `maxRPC2` and `maxRPC3`, from all the light versions of the coarse-grained algorithms, and from one of the most competitive algorithms (`maxRPC3`) in tandem with the dom/wdeg heuristics of Section 4 (`lmaxRPC3+H`). Results show that in terms of run time our algorithms have similar performance and are superior to existing ones by a factor of two on average. This is due to the elimination of many redundant constraint checks as the cc numbers show. Heuristic do not seem to make any difference.

Tables 2 and 3 compare the performance of search algorithms that apply `lmaxRPC` throughout search on RLFAPs and an indicative collection of other problems respectively. The algorithms compared are `lmaxRPC`^{rm} and `lmaxRPC3`^{rm} with and without the use of heuristic dom/wdeg for propagation list and for Case 1 of Section 4. We also include results from MAC^{rm} which is

considered the most efficient version of MAC [8, 9].

Table 2: Cpu times (t) in secs, nodes (n) and constraint checks (cc) from RLFAP instances. Algorithms that use heuristics are denoted by their name + H. The best cpu time among the lmaxRPC methods is highlighted.

instance		AC^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC^{rm} + H	lmaxRPC3^{rm} + H
scen11	t	5.4	13.2	4.6	12.5	**4.3**
	n	4,367	1,396	1,396	1,292	1,292
	cc	5M	92M	29M	90M	26M
scen11-f10	t	11.0	29.0	12.3	22.3	**9.8**
	n	9,597	2,276	2,276	1,983	1,983
	cc	11M	141M	51M	114M	41M
scen2-f25	t	**27.1**	109.2	43.0	79.6	32.6
	n	43,536	8,310	8,310	6,179	6,179
	cc	44M	427M	151M	315M	113M
scen3-f11	t	**7.4**	30.8	12.6	17.3	7.8
	n	7,962	2,309	2,309	1,852	1,852
	cc	9M	132M	46M	80M	29M
scen11-f7	t	4,606.5	8,307.5	3,062.8	6,269.0	**2,377.6**
	n	3,696,154	552,907	552,907	522,061	522,061
	cc	4,287M	35,897M	9,675M	22,899M	6,913M
scen11-f8	t	**521.1**	2,680.6	878.0	1,902.4	684.7
	n	345,877	112,719	112,719	106,352	106,352
	cc	638M	10,163M	3,172M	7,585M	2,314M
graph8-f10	t	16.4	16.8	9.1	11.0	**6.3**
	n	18,751	4,887	4,887	3,608	3,608
	cc	14M	71M	31M	51M	21M
graph14-f28	t	31.4	4.1	3.1	2.6	**2.1**
	n	57,039	2,917	2,917	1,187	1,187
	cc	13M	17M	8M	13M	6M
graph9-f9	t	273.5	206.3	**101.5**	289.5	146.9
	n	273,766	26,276	26,276	49,627	49,627
	cc	158M	729M	290M	959M	371M

Experiments showed that lmaxRPC^{rm} is the most efficient among existing algorithms when applied during search, which confirms the results given in [10]. Accordingly, lmaxRPC3^{rm} is the most efficient among our algorithms. It is between two and four times faster than maxRPC3^{rm} on hard instances, while algorithms lmaxRPC3 and lmaxRPC2 are not competitive when used during search because of the data structures they maintain. In general, when applied during search, any maxRPC algorithm is clearly inferior to the corresponding light version. The reduction in visited nodes achieved by the former is relatively small and does not compensate for the higher run times of enforcing maxRPC.

Results from Tables 2 and 3 demonstrate that lmaxRPC3^{rm} always outperforms lmaxRPC^{rm}, often considerably. This was the case in all 200 instances tried. The use of heuristics improves the performance of both lmaxRPC algo-

rithms in most cases. Looking at the columns for lmaxRPC^{rm} and lmaxRPC3^{rm} +H we can see that our methods can reduce the numbers of constraint checks by as much as one order of magnitude (e.g. in quasigroup problems qcp and qwh). This is mainly due to the elimination of redundant checks inside function *searchPCwit*. Cpu times are not cut down by as much, but a speed-up of more than 3 times can be obtained (e.g. scen2-f25 and scen11-f8).

Table 3: Cpu times (t) in secs, nodes (n) and constraint checks (cc) from various instances.

instance		AC^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC^{rm} + H	lmaxRPC3^{rm} + H
rand-2-40-8	t	**4.0**	47.3	21.7	37.0	19.0
-753-100-75	n	13,166	8,584	8,584	6,915	6,915
	cc	7M	289M	82M	207M	59M
geo50-20	t	**102.7**	347.7	177.5	273.3	150.3
d4-75-1	n	181,560	79,691	79,691	75,339	75,339
	cc	191M	2,045M	880M	1,437M	609M
qcp150-120-5	t	52.1	89.4	**50.2**	80.0	55.3
	n	233,311	100,781	100,781	84,392	84,392
	cc	27M	329M	53M	224M	36M
qcp150-120-9	t	226.8	410.7	238.1	239.9	**164.3**
	n	1,195,896	583,627	583,627	315,582	315,582
	cc	123M	1,613M	250M	718M	112M
qwh20-166-1	t	52.6	64.3	38.9	21.2	**14.9**
	n	144,653	44,934	44,934	13,696	13,696
	cc	19M	210M	23M	53M	6M
qwh20-166-6	t	1,639.0	1,493.5	867.1	1,206.2	**816.5**
	n	4,651,632	919,861	919,861	617,233	617,233
	cc	633M	5,089M	566M	3,100M	351M
qwh20-166-9	t	41.8	41.1	**25.0**	39.9	28.5
	n	121,623	32,925	32,925	26,505	26,505
	cc	15M	135M	15M	97M	11M
blackHole	t	**1.8**	14.4	3.8	12.1	3.6
4-4-e-8	n	8,661	4,371	4,371	4,325	4,325
	cc	4M	83M	12M	68M	10M
queens-100	t	**15.3**	365.3	106.7	329.8	103.0
	n	7,608	6,210	6,210	5,030	5,030
	cc	6M	1,454M	377M	1,376M	375M
queenAttacking5	t	**34.3**	153.1	56.7	136.0	54.8
	n	139,534	38,210	38,210	33,341	33,341
	cc	35M	500M	145M	436M	128M
queensKnights	t	217.0	302.0	**173.6**	482.0	283.5
-15-5-mul	n	35,445	13,462	13,462	12,560	12,560
	cc	153M	963M	387M	1,795M	869M

Importantly, the speed-ups obtained can make a search algorithm that efficiently applies lmaxRPC competitive with MAC on many instances. For instance, in scen11-f10 we achieve the same run time as MAC while lmaxRPC^{rm} is 3 times slower while in scen11-f7 we go from 2 times slower to 2 times faster. In addition,

there are several instances where MAC is outperformed (e.g. the graph RLFAPs and most quasigroup problems). Of course, there are still instances where MAC remains considerably faster despite the improvements.

Table 4: Average search performance in all 200 instances grouped by class.

Problem class		AC^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC^{rm} + H	lmaxRPC3^{rm} + H
RLFAP	t	242.8	556.7	199.3	416.3	**157.3**
(scen,graph)	cc	233M	2,306M	663M	1,580M	487M
Random	t	**8.4**	28.0	14.8	28.5	17.1
(modelB,forced)	cc	14M	161M	60M	137M	51M
Geometric	t	**21.5**	72.2	37.2	57.6	32.1
	cc	39M	418M	179M	297M	126M
Quasigroup	t	147.0	162.5	94.9	128.9	**89.6**
(qcp,qwh,bqwh)	cc	59M	562M	68M	333M	40M
QueensKnights,	t	**90.2**	505.2	180.3	496.4	198.1
Queens,QueenAttack	cc	74M	1,865M	570M	1,891M	654M
driver,blackHole	t	**3.2**	17.1	9.1	11.9	7.0
haystacks,job-shop	cc	1.8M	55M	6.4M	36.7M	5.1M

Table 4 summarizes results from the application of lmaxRPC during search. We give average results for all the tested instances, grouped into specific problem classes. As can be seen, our best method improves on the existing best one considerably, making lmaxRPC outperform MAC on the RFLAP and quasigroup problem classes. Overall, our results demonstrate that the efficient application of a maxRPC approximation throughout search can give an algorithm that is quite competitive with MAC on many binary CSPs. This confirms the conjecture of [6] about the potential of maxRPC as an alternative to AC. In addition, our results, along with ones in [10], show that approximating strong and complex local consistencies can be very beneficial.

6 Conclusion

We presented maxRPC3 and maxRPC3^{rm}, two new algorithms for maxRPC, and their light versions that approximate maxRPC. These algorithms build on and improve existing maxRPC algorithms, achieving the elimination of many redundant constraint checks. We also investigated heuristics that can be used to order certain operations within maxRPC algorithms. Experimental results from various problem classes demonstrate that our best method, lmaxRPC3^{rm}, constantly outperforms existing algorithms, often by large margins. Significantly, the speed-ups obtained allow lmaxRPC3^{rm} to compete with and outperform MAC on many problems. In the future we plan to adapt techniques for using residues from [9] to improve the

performance of our algorithms during search. Also, it would be interesting to investigate the applicability of similar methods to efficiently achieve or approximate other local consistencies.

References

[1] 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.

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

[3] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *CP-2004 Workshop on Constraint Propagation*, 2004.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI-2004*, 2004.

[5] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *CP-97*, pages 312–326, 1997.

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

[7] F. Grandoni and G. Italiano. Improved Algorithms for Max-Restricted Path Consistency. In *Proceedings of CP'03*, pages 858–862, 2003.

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

[9] C. Likitvivatanavong, Y. Zhang, J. Bowen, S. Shannon, and E. Freuder. Arc Consistency during Search. In *Proceedings of IJCAI-2007*, pages 137–142, 2007.

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

[11] R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI*, pages 163–169, Vancouver, British Columbia, Canada, 1992.