Constraints (2011) 16:372–406
DOI 10.1007/s10601-011-9110-y

New algorithms for max restricted path consistency

Thanasis Balafoutis · Anastasia Paparrizou · Kostas Stergiou · Toby Walsh

Published online: 29 June 2011

Abstract Max Restricted Path Consistency (maxRPC) is a local consistency for binary constraints that enforces a higher order of consistency than arc consistency. Despite the strong pruning that can be achieved, maxRPC is rarely used because 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 and evaluate techniques that can boost the performance of maxRPC algorithms by eliminating many of these overheads and redundancies. These include the combined use of two data structures to avoid many redundant constraint checks, and the exploitation of residues to quickly verify the existence of supports. Based on these, we propose a number of closely related maxRPC algorithms. The first one, `maxRPC3`, has optimal O(end^3) time complexity, displays good performance when used stand-alone, but is expensive to apply during search. The second one, `maxRPC3`^{rm}, has O(en^2d^4) time complexity, but a restricted version with O(end^4) complexity can be very efficient when used during search. The other algorithms are simple modifications of `maxRPC3`^{rm}. All algorithms have O(ed) space complexity when used stand-alone. However, `maxRPC3` has O(end) space

This paper is an extended version of [1] that appeared in the proceedings of CP-2010.

T. Balafoutis
Department of Information and Communication Systems Engineering,
University of the Aegean, University Hill, Administration Bldg., 81100,
Mytilene, Lesvos, Greece

A. Paparrizou · K. Stergiou (✉)
Department of Informatics and Telecommunications Engineering,
University of Western Macedonia, Karamanli & Lygeris, 50100 Kozani, FL, Greece
e-mail: kstergiou@uowm.gr

A. Paparrizou
e-mail: apaparrizou@uowm.gr

T. Walsh
NICTA, University of New South Wales, Sydney, NSW 2052, Australia

complexity when used during search, while the others retain the O(ed) complexity. Experimental results demonstrate that the resulting methods constantly outperform previous algorithms for maxRPC, often by large margins, and constitute a viable alternative to arc consistency on some problem classes.

Keywords Constraint propagation · Binary constraints · Local consistency

1 Introduction

Max Restricted Path Consistency (maxRPC) is a strong domain filtering consistency for binary constraints introduced in 1997 by Debruyne and Bessiere [14]. maxRPC achieves a stronger level of local consistency than arc consistency (AC), and in [15] 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. Fewer new algorithms have been proposed and their experimental evaluation has been very limited. In this paper we propose new algorithms for maxRPC and evaluate them empirically on a wide range of problems.

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 that is constrained with x. 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 [14], and two more algorithms have been proposed since then [18, 28]. The algorithms of [14] and [28] have been evaluated on random problems only, while the algorithm of [18] has not been experimentally evaluated at all. Despite achieving stronger pruning than AC, existing maxRPC 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 maxRPC 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 [10, 18, 22, 23, 28]. 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.

We further investigate the use of residues to improve the performance of maxRPC filtering during search. To be precise, we adapt ideas from [23] to obtain two variants of the maxRPC3^{rm} algorithm. The first one achieves a better time complexity but is inferior to maxRPC3^{rm} in practice, while the second one exploits in a simple way information obtained in the initialization phase of maxRPC3^{rm} to achieve competitive performance.

Similar algorithmic improvements can be applied to *light maxRPC* (lmaxRPC), an approximation of maxRPC [28]. This achieves a lesser level of consistency compared to maxRPC, but still stronger than AC, and is more cost-effective when used during search. Experiments confirm that lmaxRPC is indeed a considerably better option than maxRPC when used throughout search. We also propose a number of heuristics that can be used to order the searches for PC supports and witnesses during the execution of a coarse-grained maxRPC algorithm, and in this way potentially save constraint checks.

Finally, 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. We ran experiments with maxRPC algorithms under both a 2-way and a *d*-branching scheme. Results show that our methods constantly outperform existing algorithms, often by large margins, especially when 2-way branching is used. 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 three 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 some problems. Finally, we explore a simple hybrid propagation scheme where AC and maxRPC are interleaved under 2-way branching. Results demonstrate that instantiations of this scheme offer an efficient alternative to the application of a fixed propagation method (either AC or maxRPC) throughout search.

At this point we need to note that research on higher-order consistencies for binary constraints is not only motivated by the need to solve binary CSPs faster. Although there are some real problems that can be modeled using binary constraints only (e.g. frequency assignment problems), most real-life problems involve non-binary constraints. However, binary constraints still play an important role in Constraint Programming. First of all, some real problems include binary as well as non-binary constraints. Since applying higher-order consistencies on binary constraints is generally cheaper, such problems may benefit from applying a strong consistency on their binary part. Second, there are certain global constraints that can be reformulated as collections of binary or other low-arity constraints (decomposition methods), and indeed this is the approach taken by some solvers [9, 25]. Maintaining a strong local consistency on these reformulations may reduce the overall resolution time. Finally, as past experience has shown, research on binary constraints can inspire corresponding work on non-binary ones.

The remainder of this paper is structured as follows. Section 2 reviews background information on CSPs and related work on maxRPC algorithms. Section 3 presents two new algorithms, maxRPC3, maxRPC3^{rm} and their corresponding approximations, and analyzes their complexities. Section 4 discusses the further exploitation of

residues on two variations of the maxRPC3^{rm} algorithm. Section 5 discusses heuristics for (l)maxRPC algorithms, and Section 6 presents our experimental results on benchmark problems. Finally, Section 7 concludes and discusses possible directions for future work.

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, $D = \{D^0(x_1), \ldots, D^0(x_n)\}$ is a set of initial 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 $D^0(x_1) \times \ldots \times D^0(x_m)$ 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 $c_{i,j}$, and $D(x_i)$ will denote the current domain of variable x_i.

Each tuple τ of a constraint c is an ordered list of values $(a_1, \ldots, a_m)$ such that $a_j \in D^0(x_j)$, $j = 1, \ldots, m$. That is, a tuple $\tau \in rel(c)$ is a combination of values for the variables in the scope of $rel(c)$. Tuple τ *satisfies* constraint c iff $\tau \in rel(c)$. Tuple $\tau = (a_1, \ldots, a_m)$ is *valid* iff $a_j \in D(x_j)$, for $j = 1, \ldots, m$. In words, a valid tuple is an assignment of values to the variables involved in the constraint such that none of these values has been removed from the domain of the corresponding variable.

The process which verifies whether a given tuple is allowed by (i.e. satisfies) a constraint c is called a *constraint check*. A constraint c can be either defined *extensionally* by explicitly giving $rel(c)$, or (usually) *intensionally* by implicitly specifying $rel(c)$ through a predicate or arithmetic function. A binary CSP is a CSP where each constraint involves at most two variables and is typically represented by a constraint graph where nodes correspond to variables and edges correspond to constraints. 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 $c_{i,j}$ there exists a value $a_j \in D(x_j)$ s.t. the pair of values (a_i, a_j) satisfies $c_{i,j}$. 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.

Complete algorithms for CSPs are based on exhaustive backtracking search interleaved with constraint propagation. Search is typically guided by variable and value ordering heuristics and makes use of a specific branching scheme like 2-way or d-way branching.

In the experiments presented in this paper we have used the *dom/wdeg* variable ordering heuristic which is one of the most efficient general-purpose heuristics for CSPs [12]. This heuristic assigns a weight to each constraint, initially set to one. Each variable is associated with a *weighted degree* (wdeg), which is the sum of the weights over all constraints involving the variable and at least another (unassigned) variable. The *dom/wdeg* heuristic chooses the variable with minimum ratio of current domain size to weighted degree.

The two most widely used branching schemes are 2-way and d-way. In 2-way branching, after a variable x is chosen and a value $a_i \in D(x)$ is selected, two branches are created [26]. In the left branch a_i is assigned to x, namely the constraint $x = a_i$ is added to the problem and is propagated. In the right branch the constraint $x \neq a_i$ is

added to the problem and is propagated. If the left branch fails and the propagation of $x \neq a_i$ succeeds then any variable can be selected next (not necessarily x). If both branches fail then the algorithm backtracks. In d-way branching, after variable x is selected, d branches are built, each one corresponding to one of the d possible value assignments of x. If the branch corresponding to assignment $x = a_i$ fails, the next available value assignment to x is tried (next branch), and so on. If all d branches fail then the algorithm backtracks. In Section 6 we report results from experiments with both branching schemes.

2.1 maxRPC

Several local consistencies stronger than AC have been proposed in the literature. Domain filtering consistencies are especially interesting because they only filter values from domains and as a result they do not alter either the constraint graph by adding new constraints, or the constraints' relations by removing inconsistent tuples [15]. Examples of domain filtering consistencies for binary constraints include restricted path consistency [4], path inverse consistency and neighborhood inverse consistency [16], max restricted path consistency [14], and singleton arc consistency [15]. The most famous local consistency that is not cast as domain filtering is path consistency. When enforced, path consistency can remove inconsistent 2-tuples from binary relations and/or introduce new binary constraints. Another recent example of such a local consistency is dual consistency [21].

The detailed study of domain filtering consistencies given in [15] concluded that maxRPC and SAC are promising strong consistencies that could be viable alternatives to AC. SAC, which is stronger than maxRPC, has received considerable attention since, and significant progress has been made regarding new SAC algorithms [3, 6, 7, 20]. However, there is no concrete evidence yet that SAC can be cost-effective when maintained during search.

maxRPC was introduced in [14] as an extension to *Restricted Path Consistency* (RPC). A binary CSP is RPC iff it is AC and for each constraint $c_{i,j}$ and each value $a_i \in D(x_i)$ that has a single AC-support $a_j \in D(x_j)$, the pair of values (a_i, a_j) is *path consistent* (PC) [4]. 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 we say that 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 value $a_i \in D(x_i)$ is *max restricted path consistent* (maxRPC) iff it is AC and for each constraint $c_{i,j}$ 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 [14]. A variable is maxRPC iff all its values are maxRPC. A problem is maxRPC iff there is no empty domain in D and all variables in X are maxRPC.

A local consistency related to maxRPC is *Path Inverse Consistency* (PIC) [16]. A value $a_i \in D(x_i)$ is PIC iff for all $x_j, x_k \in X$, s.t. $x_i \neq x_j \neq x_k \neq x_i$, there exist $b_j \in D(x_j)$ and $d_k \in D(x_k)$, s.t. the assignments (x_i, a_i), (x_j, b_j) and (x_k, d_k) satisfy the constraints between the three variables. Less formally, a_i is PIC iff it is AC and it can be extended to all the 3-cliques of variables containing x_i. A variable is PIC iff all its values are PIC. A problem is PIC iff there is no empty domain in D and all variables in X are PIC.

A theoretical analysis and experimental results presented in [15] demonstrated that maxRPC is more efficient compared to RPC and PIC. Hence it was identified as

a promising alternative to AC. This is why we focus on this local consistency in this paper.

Three algorithms for achieving maxRPC have been proposed in the literature so far. The first one, called maxRPC1 [14], is a fine-grained algorithm based on AC-6 [5] and has optimal O(end^3) time complexity and O(end) space complexity. The second algorithm, called maxRPC2 [18], is a coarse-grained algorithm that uses ideas similar to those used by the AC algorithm AC2001/3.1 to achieve O(end^3) time and O(ed) space complexity at the cost of some redundant checks compared to maxRPC1. The third algorithm, maxRPC^{rm} [28], is a coarse-grained algorithm based on AC3^{rm} [22]. The time and space complexities of maxRPC^{rm} are O(en^2d^4) and O(end). Note that in [28] the complexities are given as O($eg + ed^3 + csd^4$) and O($ed + cd$), where c is the number of 3-cliques, g is the maximum degree of a variable and s is the maximum number of 3-cliques that share the same single constraint in the constraint graph. Considering that c is O(en) and s is O(n), we can derive the complexities for maxRPC^{rm} given here. This algorithm has a higher time complexity than the other two, but it has some advantages compared to them because of its lighter use of data structures during search (this is explained below and in Section 3.2). Finally, maxRPCEn1 is a fine-grained algorithm closely related to maxRPC1 [13]. This algorithm is based on AC-7 [8] and achieves maxRPCEn, a local consistency stronger than maxRPC.

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 as explained below.

Central to maxRPC2 is the *Last PC* data structure, as we call it here. For each constraint $c_{i,j}$ and each value $a_i \in D(x_i)$, $Last\,PC_{x_i,a_i,x_j}$ gives the most recently discovered PC-support of a_i in $D(x_j)$. maxRPC2 maintains this data structure incrementally. This means that a copy of *Last PC* is made when moving forward during search (i.e. after a successfully propagated variable assignment) and *Last PC* is restored to its previous state after a failed variable assignment (or value removal in the case of 2-way branching). Data structures with such a property are often referred to as *backtrackable*. Their use implies an increased space complexity as copies need to be made while search progresses down a branch. On a brighter note, since *Last PC* is backtrackable, maxRPC2 displays the following behavior: When looking for a PC-support for a_i in $D(x_j)$, it first checks if $Last\,PC_{x_i,a_i,x_j}$ is valid. If it is not, it searches for a new PC-support starting from the value *immediately after* $Last\,PC_{x_i,a_i,x_j}$ 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 *Last PC* to store *residual supports* or simply *residues*, i.e. supports that have been discovered during execution and stored for future use, but does not maintain this structure incrementally (it only needs one copy). Therefore, no additional actions need to be taken (copying or restoration) when moving forward or after a fail. Such data structures are often referred to as *backtrack-stable*. When looking for a PC-support for a_i in $D(x_j)$, if the residue $Last\,PC_{x_i,a_i,x_j}$ is not valid then maxRPC^{rm} searches for a new PC-support from scratch in $D(x_j)$. This results in higher time complexity, but crucially does not require costly maintainance of *Last PC* during search. The algorithm also makes use of residues for the PC-witnesses found in every third variable for each

pair (a_i, a_j). These are stored in a data structure with an O(*end*) space complexity. The initialization of this structure causes an extra overhead which can be significant on very large problems.

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 [18] and [28]. Both algorithms use a propagation list Q where variables whose domain is pruned are added. Once a variable x_j is removed from Q 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 three possible reasons for deletion:

- The first is when a_i no longer has an AC-support in $D(x_j)$.
- The second, 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 third, 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.

If any of the above cases occurs then value a_i is no longer maxRPC.

We now present the pseudocodes for the new maxRPC algorithms, maxRPC3 and maxRPC3^{rm}. Both algorithms utilize data structures $Last\,PC$ and $Last\,AC$ which have the following functionalities: For each constraint $c_{i,j}$ and each value $a_i \in D(x_i)$, $Last\,PC_{x_i,a_i,x_j}$ and $Last\,AC_{x_i,a_i,x_j}$ point to the most recently discovered PC and AC supports of a_i in $D(x_j)$ respectively. Initially, all $Last\,PC$ and $Last\,AC$ pointers are set to a special value NIL, considered to precede all values in any domain. As will be explained, algorithm maxRPC3 updates the $Last\,PC$ and $Last\,AC$ structures incrementally like maxRPC2 and AC2001/3.1 respectively do. In contrast, algorithm maxRPC3^{rm} uses these structures as residues like maxRPC^{rm} and AC3^{rm} do.

3.1 maxRPC3

The main part of maxRPC3 is described in Algorithm 1. Since maxRPC3 is coarse-grained, it uses a propagation list Q (typically implemented as a queue) where variables that have their domain filtered are inserted. This may happen during initialization (explained below) or when PC-support or PC-witness loss is detected. When a variable x_j is removed from Q, at line 4, each variable x_i constrained with x_j

must be checked for possible AC-support, PC-support or PC-witness loss. We now discuss the overall function of the algorithm before moving on to explain it in detail.

For each value $a_i \in D(x_i)$, Algorithm 1 first checks whether a_i has suffered AC-support or PC-support loss in $D(x_j)$ by calling function *checkPCsupLoss*, provided that $LastPC_{x_i,a_i,x_j}$ is not valid anymore (line 7). This function, which will be explained in detail below, returns *false* if no new PC-support exists for a_i in $D(x_j)$ and as a result a_i is deleted (line 8). If a_i is not deleted, either because $LastPC_{x_i,a_i,x_j}$ is still valid or because a new PC-support for a_i has been found in $D(x_j)$, then possible PC-witness loss is examined by calling function *checkPCwitLoss* (line 11). If this function returns *false*, then a_i is deleted (line 12). If a value is deleted from $D(x_i)$ then x_i is inserted to Q (lines 9 and 13). After deleting values from the domain of a variable, the algorithm checks whether the domain is empty (line 14). If so, the algorithm returns FAILURE.

An important remark about Algorithm 1 is the following. Assuming a value a_i has been examined in lines 6–13 and has not been deleted, then this does not necessarily mean that a_i is maxRPC. Indeed there is the possibility that $LastPC_{x_i,a_i,x_j}$ is valid but the last PC-witness of the pair $(a_i, LastPC_{x_i,a_i,x_j})$ in some variable x_k has been deleted. Hence, if $LastPC_{x_i,a_i,x_j}$ is the last PC-support of a_i in $D(x_j)$ then a_i is not maxRPC. Such a situation will be identified at some point during the execution of the algorithm once x_k is removed from Q and its neighboring variables are examined. This guarantees the algorithm's completeness as will be further explained in Section 3.4.

The initialization step of `maxRPC3` (Function 2) is a brute-force function, where each value a_i of each variable x_i is checked for being maxRPC. This is done by iterating through the variables constrained with x_i and looking for a PC-support for a_i in their domains. For each such variable x_j and value $a_j \in D(x_j)$, we first check if the pair (a_i, a_j) is arc consistent by calling function *isConsistent* at line 6. *isConsistent* returns *true* if (a_i, a_j) satisfies the constraint, meaning that a_j AC-supports a_i. In this case $LastAC_{x_i,a_i,x_j}$ is set to a_j (line 7). If a_j is verified as an AC-support of a_i, we examine if it is also a PC-support by calling function *searchPCwit*. If *searchPCwit* returns *true* (detailed analysis follows below), then $LastPC_{x_i,a_i,x_j}$ is set to a_j (line 10), since a_j is the most recently found PC-support for a_i. Line 11 will be explained below when algorithm `maxRPC3`^{rm} is presented. Then, the next variable constrained with x_i will be considered, and so on.

Algorithm 1 maxRPC3

```
1: if ¬ initialization(Q, LastPC, LastAC) then
2:     return FAILURE;
3: while Q ≠ ∅ do
4:     Q ← Q−{x_j};
5:     for each x_i ∈ X s.t. c_{i,j} ∈ C do
6:         for each a_i ∈ D(x_i) do
7:             if LastPC_{x_i,a_i,x_j} ∉ D(x_j) AND ¬ checkPCsupLoss(a_i, x_j) then
8:                 remove a_i;
9:                 Q ← Q ∪ {x_i};
10:            else
11:                if ¬ checkPCwitLoss(x_i, a_i, x_j) then
12:                    remove a_i;
13:                    Q ← Q ∪ {x_i};
14:        if D(x_i) = ∅ then
15:            return FAILURE;
16: return SUCCESS;
```

Function 2 initialization(Q, $LastPC$, $LastAC$):**boolean**

```
1: for each x_i ∈ X do
2:     for each a_i ∈ D(x_i) do
3:         for each x_j ∈ X s.t. c_{i,j} ∈ C do
4:             maxRPCsupport ← FALSE;
5:             for each a_j ∈ D(x_j) do
6:                 if isConsistent(a_i, a_j) then
7:                     LastAC_{x_i,a_i,x_j} ← a_j;
8:                     if searchPCwit(a_i, a_j) then
9:                         maxRPCsupport ← TRUE;
10:                        LastPC_{x_i,a_i,x_j} ← a_j;
11:                        if (rm) then LastPC_{x_j,a_j,x_i} ← a_i;
12:                        break;
13:            if ¬ maxRPCsupport then
14:                remove a_i;
15:                Q ← Q ∪ {x_i};
16:                break;
17:    if D(x_i) = ∅ then
18:        return FALSE;
19: return TRUE;
```

If there is no AC-support in $D(x_j)$ for a_i or none of the AC-supports is a PC-support, then a_i will be removed at line 14 and x_i will be added to queue Q. Eventually, a_i is established to be maxRPC when a PC-support is found in each $D(x_j)$, where x_j has a constraint with x_i. Finally, if function *initialization* causes an empty domain (line 17), then `maxRPC3` returs FAILURE in line 2 of Algorithm 1. Note that *initilization* is called only when `maxRPC3` is used stand-alone (e.g. for preprocessing) and not during search, as in this case Q is initialized with the variable of the latest decision.

Assuming the initialization phase succeeded, the propagation list Q will include those variables that have their domain filtered. The main part of `maxRPC3` (Algorithm 1) starts when a variable x_j is extracted from Q (line 4) in order to determine whether a neighbouring variable (x_i) has suffered PC-support or PC-witness loss due to the filtering of the extracted variable's domain. These checks are implemented by calling functions *checkPCsupLoss* and *checkPCwitLoss*, at lines 7 and 11 of Algorithm 1, for each value $a_i \in D(x_i)$. If value $LastPC_{x_i,a_i,x_j}$ is still in $D(x_j)$ line 7, then a *possible* PC-support has been immediately located (the PC-support will be established later as explained in the remark about the algorithm given above) and *checkPCsupLoss* is not called. In the opposite case where $LastPC_{x_i,a_i,x_j}$ is not valid, *checkPCsupLoss* is called to search for a new PC-support in $D(x_j)$.

3.1.1 Checking for PC-support loss

Function *checkPCsupLoss* (Function 3) takes advantage of the $LastPC$ and $LastAC$ pointers to avoid starting the search for PC-support 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 1–4 in *checkPCsupLoss* 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 $LastPC_{x_i,a_i,x_j}$ and thus may perform redundant constraint checks.

For every value $a_j \in D(x_j)$, starting with b_j, we first check if it is an AC-support of a_i by calling function *isConsistent* (line 6). If it is, then we can update $LastAC_{x_i,a_i,x_j}$

Function 3 checkPCsupLoss(a_i, x_j):**boolean**

```
1: if LastAC_{x_i,a_i,x_j} ∈ D(x_j) then
2:     b_j ← max(LastPC_{x_i,a_i,x_j}+1, LastAC_{x_i,a_i,x_j});
3: else
4:     b_j ← max(LastPC_{x_i,a_i,x_j}+1, LastAC_{x_i,a_i,x_j} + 1);
5: for each a_j ∈ D(x_j), a_j ≥ b_j do
6:     if isConsistent(a_i, a_j) then
7:         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
8:             LastAC_{x_i,a_i,x_j} ← a_j;
9:         if searchPCwit(a_i, a_j) then
10:            LastPC_{x_i,a_i,x_j} ← a_j;
11:            return TRUE;
12: return FALSE;
```

under a certain condition (lines 7–8). 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 update, 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* (line 9). If no PC-support for a_i is found in $D(x_j)$, *checkPCsupLoss* will return *false*, a_i will be deleted and x_i will be added to Q in Algorithm 1. Otherwise, $LastPC_{x_i,a_i,x_j}$ is set to the discovered PC-support a_j (line 10).

Function *searchPCwit* (Function 4) 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.[1] First, taking advantage of the *LastAC* pointers, it makes a quick check in constant time which, if successful, can save searching in the domain of x_k. To be precise, it checks if $LastAC_{x_i,a_i,x_k}$ is valid and $LastAC_{x_i,a_i,x_k}$ equals $LastAC_{x_j,a_j,x_k}$, or if $LastAC_{x_i,a_i,x_k}$ is valid and consistent with a_j or if $LastAC_{x_j,a_j,x_k}$ is valid and consistent with a_i (line 3). The first part of the disjunction is of practical importance only, since if it is true, then the second part will necessarily also be true and the condition will be verified. However, including the first part of the condition saves constraint checks, and this reflects on run times in certain problems.

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 check as they do not have the *LastAC* structure. In contrast, algorithm `maxRPCEn1` is able to do such reasoning. Experimental results in Section 6 demonstrate that these simple conditions of line 3 can eliminate a very large number of redundant constraint checks.

If none of the conditions in line 3 of Function 4 holds, searching for a new PC-witness in $D(x_k)$ is necessary. This is done by first calling function *searchACsup* (Function 5), first with (a_i, x_k) and then with (a_j, x_k) as parameters. This function locates the lexicographically smallest AC-supports for a_i and a_j in $D(x_k)$. More precisely, *searchACsup* checks if the current *LastAC* value exists in the corresponding domain (line 1 of Function 5), and if not it searches for a new AC-support after that (line 4). If it finds one, it updates *LastAC* accordingly (line 6).

[1] 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 [24].

Function 4 searchPCwit(a_i, a_j):**boolean**

```
1: for each x_k ∈ X s.t. c_{i,k} and c_{j,k} ∈ C do
2:     maxRPCsupport ← FALSE;
3:     if ((LastAC_{x_i,a_i,x_k} ∈ D(x_k)) AND (LastAC_{x_i,a_i,x_k}=LastAC_{x_j,a_j,x_k})) OR ((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
4:         continue;
5:     if ¬searchACsup(a_i, x_k) OR ¬searchACsup(a_j, x_k) then
6:         return FALSE;
7:     for each a_k ∈ D(x_k), a_k ≥ max(LastAC_{x_i,a_i,x_k}, LastAC_{x_j,a_j,x_k}) do
8:         if isConsistent(a_i, a_k) AND isConsistent(a_j, a_k) then
9:             maxRPCsupport ← TRUE;
10:            break;
11:    if ¬ maxRPCsupport then
12:        return FALSE;
13: return TRUE;
```

Then, going back to *searchPCwit* the search for a PC-witness starts from $b_k = max\{LastAC_{x_i,a_i,x_k}, LastAC_{x_j,a_j,x_k}\}$ (line 7), exploiting the $LastAC$ structure to save redundant checks (a similar operation is performed by `maxRPCEn1`). This search looks for a value of x_k that is compatible with both a_i and a_j (line 8). If no AC-support is found for either a_i or a_j (in which cases *searchACsup* returns *false*) or no PC-witness is located, then subsequently *searchPCwit* will also return *false*.

3.1.2 Checking for PC-witness loss

In `maxRPC3`, if value a_i is not removed after checking for possible PC-support loss using *checkPCsupLoss*, function *checkPCwitLoss* (Function 6) 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 verify if there is still a PC-witness in $D(x_j)$. As in function *searchPCwit*, taking advantage of the $LastAC$ pointers, we first make a quick check in constant time which, if successful, can save searching in the domain of x_j. That is, we check if $LastAC_{x_i,a_i,x_j}$ is valid and $LastAC_{x_i,a_i,x_j}$ equals $LastAC_{x_k,a_k,x_j}$ or if $LastAC_{x_i,a_i,x_j}$ is valid and consistent with a_k or if $LastAC_{x_k,a_k,x_j}$ is valid and consistent with a_i (line 4). If none of these conditions holds then we search for a PC-witness starting from $b_j = max\{LastAC_{x_i,a_i,x_j}, LastAC_{x_k,a_k,x_j}\}$ (line 8), after checking the existence of AC-supports for a_i and a_k in $D(x_j)$, by calling *searchACsup* (line 7). Right here the procedure is quite similar to *searchPCwit*. If there is no AC-support in $D(x_j)$ for either a_i or a_k we avoid searching for a PC-witness in $D(x_j)$ and move on

Function 5 searchACsup(a_i, x_j):**boolean**

```
1: if LastAC_{x_i,a_i,x_j} ∈ D(x_j) then
2:     return TRUE;
3: else
4:     for each a_j ∈ D(x_j), a_j > LastAC_{x_i,a_i,x_j} do
5:         if isConsistent(a_i, a_j) then
6:             LastAC_{x_i,a_i,x_j} ← a_j;
7:             return TRUE;
8: return FALSE;
```

Function 6 checkPCwitLoss(x_i, a_i, x_j):**boolean**

```
1: for each x_k ∈ X s.t. c_{i,k} and c_{k,j} ∈ C do
2:     witness ← 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 (LastAC_{x_i,a_i,x_j}=LastAC_{x_k,a_k,x_j})) OR ((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 searchACsup(x_i, a_i, x_j) AND searchACsup(x_k, a_k, x_j) then
8:                 for each a_j ∈ D(x_j), a_j ≥ max(LastAC_{x_i,a_i,x_j}, LastAC_{x_k,a_k,x_j}) do
9:                     if isConsistent(a_i, a_j) AND isConsistent(a_k, a_j) then
10:                        witness ← TRUE;
11:                        break;
12:    if ¬ witness AND ¬ checkPCsupLoss(a_i, x_k) then
13:        return FALSE;
14: return TRUE;
```

to seek a new PC-support for a_i in $D(x_k)$. 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 $D(x_j)$.

If $LastPC_{x_i,a_i,x_k}$ has been removed or the pair (a_i, a_k) has no PC-witness in $D(x_j)$, we search for a new PC-support for a_i in $D(x_k)$ in line 12 by calling function *checkPCsupLoss*. 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 1–4 in Function 3). The procedure was explained above when describing *checkPCsupLoss*. If the search for a PC-support fails for any third variable x_k then *false* will be returned, and in the main algorithm a_i will be deleted and x_i will be added to Q.

maxRPC3 terminates when Q becomes empty, meaning that all values are maxRPC, or, when a domain of some variable becomes empty, meaning that the problem is not consistent.

As observed above, when maxRPC3 is applied during search, the propagation list Q is initialized with the variable at the current decision (assignment or value removal). If propagating a decision results in an empty domain, then both the $LastAC$ and $LastPC$ data structures must be restored to their state prior to the decision.

3.2 maxRPC3^{rm}

maxRPC3^{rm} is a coarse-grained maxRPC algorithm that exploits backtrack-stable data structures inspired from AC3^{rm} (rm stands for multidirectional residues). $LastAC$ and $LastPC$ are not maintained incrementally as in maxRPC3, but are only used to store residues. As explained, a residue is a support which has been located and stored during the execution of the procedure that proves that a given value is AC or PC. The algorithm stores the most recently discovered AC (resp. PC) supports, but does not guarantee that any lexicographically smaller value is not an AC (resp. PC) support. Consequently, when we search for a new AC or PC support in a domain, we always start from scratch. $LastAC$ and $LastPC$ need not be restored after a failure; they can remain unchanged, hence a minimal overhead on the management of data.

Another difference with maxRPC3 is that since maxRPC3^{rm} handles $LastPC$ only as a residue, it can exploit the bidirectionality of support. This means that when a

Function 7 checkPCsupLoss$^{rm}(a_i, x_j)$:**boolean**

```
1: for each a_j ∈ D(x_j) do
2:     if isConsistent(a_i, a_j) then
3:         if searchPCwit^rm(a_i, a_j) then
4:             LastPC_{x_i,a_i,x_j} ← LastAC_{x_i,a_i,x_j} ← a_j;
5:             LastPC_{x_j,a_j,x_i} ← a_i;
6:             return TRUE;
7: return FALSE;
```

PC-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 for a_j. As a result, we can assign $LastPC_{x_i,a_i,x_j}$ and $LastPC_{x_j,a_j,x_i}$ to a_j and to a_i respectively. Although the property of bidirectionality obviously also holds for AC-supports, we do not exploit this since experiments demonstrated that it does not offer any benefits in most cases. Moreover, $LastAC$ is updated when a PC-support is found, since it is also the most recent AC-support found. This assignment may speed up subsequent searches for PC-witness as the conditions in line 3 of *searchPCwit*^{rm} and line 4 of *checkPCwitLoss*^{rm} are more likely to be *true*.

We omit presenting the main algorithm for maxRPC3^{rm} as it is the same as Algorithm 1 with the only difference being that we call *checkPCsupLoss*^{rm} and *checkPCwitLoss*^{rm} instead of *checkPCsupLoss* and *checkPCwitLoss* respectively. When maxRPC3^{rm} is used for preprocessing, the *initialization* function (Function 2) is called to initialize Q and structures $LastAC$ and $LastPC$. The difference with maxRPC3 concerns the bidirectionality of PC-supports. If the auxilary boolean variable *rm* is *true*, denoting the use of maxRPC3^{rm} instead of maxRPC3, we initialize the $LastPC$ residue exploiting bidirectionality. To be precise, when a PC-support is found for a_i in $D(x_j)$ we set $LastPC_{x_i,a_i,x_j}$ to a_j and additionally $LastPC_{x_j,a_j,x_i}$ to a_i (line 11 of Function 2).

When a variable is extracted from Q, we first explore the case of PC-support loss by calling function *checkPCsupLoss*^{rm}, after verifying that value $LastPC_{x_i,a_i,x_j}$ is not in $D(x_j)$ anymore. *checkPCsupLoss*^{rm} (Function 7) searches for a new PC-support starting from scratch (line 1). In contrast, maxRPC3 would start from $b_j = max(LastPC(x_i, a_i, x_j), LastAC(x_i, a_i, x_j))$ and maxRPC2 from the value after $LastPC_{x_i,a_i,x_j}$. When an AC-support a_j is confirmed from *isConsistent* in line 2,

Function 8 searchPCwit$^{rm}(a_i, a_j)$:**boolean**

```
1: for each x_k ∈ X s.t. c_{i,k} and c_{j,k} ∈ C do
2:     maxRPCsupport ← FALSE;
3:     if ((LastAC_{x_i,a_i,x_k} ∈ D(x_k)) AND (LastAC_{x_i,a_i,x_k} = LastAC_{x_j,a_j,x_k})) OR ((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
4:         continue;
5:     for each a_k ∈ D(x_k) do
6:         if isConsistent(a_i, a_k) AND isConsistent(a_j, a_k) then
7:             maxRPCsupport ← TRUE;
8:             LastAC_{x_i,a_i,x_k} ← LastAC_{x_j,a_j,x_k} ← a_k;
9:             break;
10:    if ¬ maxRPCsupport then
11:        return FALSE;
12: return TRUE;
```

Function 9 checkPCwitLoss$^{rm}(a_i, x_j)$:**boolean**

```
1: for each x_k ∈ X s.t. c_{i,k} and c_{k,j} ∈ C do
2:     witness ← 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 (LastAC_{x_i,a_i,x_j} = LastAC_{x_k,a_k,x_j})) OR ((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:             for each a_j ∈ D(x_j) do
8:                 if isConsistent(a_i, a_j) AND isConsistent(a_k, a_j) then
9:                     LastAC(x_i, a_i, x_j) ← LastAC(x_k, a_k, x_j) ← a_j;
10:                    witness ← TRUE;
11:                    break;
12:    if ¬ witness AND ¬ checkPCsupLoss^{rm}(a_i, x_k) then
13:        return FALSE;
14: return TRUE;
```

function *searchPCwit*^{rm} is called to determine if a_j is also a PC-support for a_i. If *searchPCwit*^{rm} returns *true*, we assign $LastPC_{x_i,a_i,x_j}$ and $LastPC_{x_j,a_j,x_i}$ to a_j and to a_i respectively to exploit bidirectionality, and $LastAC_{x_i,a_i,x_j}$ is set to a_j (lines 4–5), since the discovered PC-support is also an AC-support.

Function *searchPCwit*^{rm} (Function 8) checks if a pair of values (a_i,a_j) is PC by iterating over the variables x_k constrained with x_i and x_j. First, it checks the same conditions in line 3 as *searchPCwit* to locate, if possible, a PC-witness without searching. If none of these conditions holds, it searches for a new PC-support starting from the first value in $D(x_k)$ (line 5). If a PC-witness a_k is found (line 7) 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 (line 8). If no PC-witness is found we have determined that the pair (a_i,a_j) is not PC and as a result *false* will be returned and *checkPCsupLoss*^{rm} will move to check if the next available value in $D(x_j)$ is a PC-support for a_i.

In maxRPC3^{rm}, if value a_i is not removed after checking for possible PC-support loss using *checkPCsupLoss*^{rm}, function *checkPCwitLoss*^{rm} (Function 9) is called to check for PC-witness loss. This is done by iterating again, over the variables that are constrained with both x_i and x_j. For each such variable x_k, we first check if $a_k = LastPC_{x_i,a_i,x_k}$ remains in $D(x_k)$ (line 3) and if so, if any of the three conditions in line 4 is satisfied in order to avoid searching. In case each of these conditions fails, we search for a new PC-witness in $D(x_j)$ starting from the first value (line 7). For each value $a_j \in D(x_j)$, *checkPCwitLoss*^{rm} checks if it is compatible with a_i and a_k and moves the *LastAC* pointers accordingly (line 9), exploiting the bidirectionality of residues.

If $LastPC_{x_i,a_i,x_k}$ is not valid or the pair (a_i, a_k) fails to find a PC-witness in $D(x_j)$, we search for a new PC-support for a_i in $D(x_k)$ in line 12, by calling *checkPCsupLoss*^{rm}. If the search for a PC-support fails then *false* will be returned (line 13), a_i will be deleted, and x_i will be added to Q in the main algorithm.

3.3 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 [28]. That is, when removing a

variable x_j from Q, for each $a_i \in D(x_i)$, where x_i is constrained with x_j, lmaxRPC only checks if there is a PC-support of a_i in $D(x_j)$. 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. Hence when applying this consistency a fixed point is dependent on the particularities of the specific algorithm used, like the order in which the algorithm processes revisions of variables/constraints, and the order in which values are processed and supports as seeked. Light versions of algorithms `maxRPC3` and `maxRPC3`^{rm}, simply noted `lmaxRPC3` and `lmaxRPC3`^{rm} respectively, can be obtained by omitting the call to the *checkPCwitLoss* (resp. *checkPCwitLoss*^{rm}) function (lines 10–13 of Algorithm 1). In a similar way, we can obtain light versions of algorithms `maxRPC2` and `maxRPC`^{rm}.

As already noted in [28], 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 of the data structures described here can achieve more pruning than algorithms `lmaxRPC2`, `lmaxRPC3`, `lmaxRPC`^{rm}, and `lmaxRPC3`^{rm}, albeit being much slower in practice. This is because when looking for a PC-support for a value $a_i \in D(x_i)$ in a variable x_j, the brute-force algorithm will always search in $D(x_j)$ from scratch. In contrast, consider that any of the four more sophisticated algorithms will return *true* in case $Last\,PC_{x_i,a_i,x_j}$ is valid. However, although $a_j = Last\,PC_{x_i,a_i,x_j}$ is valid, it may no longer be a PC-support because the PC-witness for the pair (a_i, a_j) in some third variable may have been deleted, and it may be the last one. In a case where a_j was the last PC-support in x_j for value a_i, the four 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 $O(end^4)$ 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 [28]. 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.

3.4 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 14 Function 2) or in line 8 of Algorithm 1, after *checkPCsupLoss* has returned *false*, or in line 12, after *checkPCsupLoss* has returned *true* and *checkPCwitLoss* 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 *checkPCsupLoss* returns *false*, as long as $LastPC_{x_i,a_i,x_j}$ is not valid in Algorithm 1, a new PC-support in $D(x_j)$ is sought (lines 5–11 in Function 3). 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). *checkPCsupLoss* will return false either because no AC-support for a_i can be found in $D(x_j)$, or because for any AC-support found, *searchPCwit* returned false. 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($LastAC_{x_i,a_i,x_k}$, $LastAC_{x_j,a_j,x_k}$). 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 *searchACsup*, 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 *checkPCsupLoss* 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 $LastPC_{x_i,a_i,x_j}$ is valid in Algorithm 1 and a_i was removed after *checkPCwitLoss* 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 (line 12) in Function 6 of *checkPCwitLoss* 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($LastAC_{x_i,a_i,x_j}$, $LastAC_{x_j,a_j,x_j}$), after both *LastAC* pointers have been updated by *searchACsup*. 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 when calling *checkPCsupLoss(a_i,x_k)*, 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 earlier 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. Since $Last PC_{x_i,a_i,x_j}$ is no longer valid function *checkPCsupLoss* will be called to 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 $Last PC_{x_i,a_i,x_j}$ 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, *checkPCsupLoss* will find out that there is no PC-support for a_i in $D(x_j)$ and will delete it. If $Last PC_{x_i,a_i,x_j}$ is valid then *checkPCsupLoss* will be omitted (line 7 of Algorithm 1). 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 *checkPCsupLoss* and a_i will be deleted. Otherwise, function *checkPCwitLoss* 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 $Last PC_{x_i,a_i,x_j}$ is valid, a new PC-witness for (a_i,a_j) in $D(x_k)$ will be sought (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 sought (line 12). Since a_j was the last PC-support for a_i in $D(x_j)$, none will be found, *checkPCwitLoss* 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 *Last AC* and *Last PC* structures. As `maxRPC3`^{rm} does not maintain these structures incrementally, the searches for PC-supports in *checkPCsupLoss*^{rm} and *checkPCwitLoss*^{rm} and the searches for PC-witnesses in *searchPCwit*^{rm} and *checkPCwitLoss*^{rm} 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.[2] Following [23],

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

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 decision has been made (e.g. a variable assignment or a value removal). 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 *checkPCsupLoss*, *checkPCwitLoss*, and mainly *searchPCwit* where most checks are executed.

Each variable can be inserted and extracted from Q 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 Q, *checkPCsupLoss* 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 6 in *checkPCsupLoss*) and then calling *searchPCwit* (line 9). The cost of *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 Q, 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, *checkPCsupLoss* 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 *checkPCsupLoss* will be $O(end^3)$. This is the node complexity of `lmaxRPC3`.

In *checkPCwitLoss* the algorithm iterates 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 Q $O(d)$ 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 *checkPCwitLoss*, 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 sought through *searchPCwit*. This costs $O(nd^2)$ as explained above but it is amortized with the cost incurred by the calls to *searchPCwit* from *checkPCsupLoss*. Therefore, the cost of *checkPCwitLoss* 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 *checkPCsupLoss*^{rm} and *searchPCwit*^{rm}. Hence, the node complexity of lmaxRPC3^{rm} is O(end^4). Also, the total cost of *searchPCwit*^{rm} in one node cannot be amortized. This means that the cost of *searchPCwit*^{rm} when called within *checkPCwitLoss*^{rm} 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*^{rm} 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} avoids this, its branch space complexity is O(ed).

4 Further exploitation of residues in maxRPC algorithms

As detailed above, the use of the *LastPC* and *LastAC* data structures by algorithms such as maxRPC2, maxRPC3, and AC2001/3.1 can give optimal time complexity bounds. However, the overhead for maintaining the required data structures during search can outweigh the benefit of the optimal theoretical results. On the other hand, the use the *LastPC* and *LastAC* structures as residues by algorithms such as maxRPC^{rm}, maxRPC3^{rm}, and AC3^{rm} sacrifices the optimal time complexity to achieve better average performance in practice.[3]

In this section we investigate variants of maxRPC3^{rm} that offer a compromise between maxRPC3^{rm} and maxRPC3 by exploring ideas presented in [23] regarding the use of residues in AC algorithms. The first variant of maxRPC3^{rm}, called maxRPC3-resOpt, uses an extra data structure to record the current PC-supports before the invocation of the maxRPC algorithm at each node of the search tree. As explained below, by exploiting this data structure we can achieve an improved node time complexity. The second variant, called maxRPC3-start, also introduces an additional data structure, but only makes use of information obtained during the initialization phase of the maxRPC algorithm. This does not improve the asymptotic time complexity, but results in better average performance in practice.

4.1 maxRPC3-resOpt

Algorithm maxRPC3-resOpt is inspired from the ACS-resOpt algorithm of [23]. Adapting the main idea of ACS-resOpt to maxRPC, we use a data structure, called *Stop*, to copy and remember the residues in *LastPC* each time a node is visited right before the maxRPC algorithm is invoked. Also, we view each domain as being "circular". That is, the last value in the initial domain of a variable is followed

[3]This is verified by experimental results given in [22, 23, 28] and also in Section 6 here.

Function 10 checkPCsupLoss-resOpt(a_i, x_j):**boolean**

1: $a_j \leftarrow Last PC_{x_i,a_i,x_j}$+1;
2: **while** $a_j \neq \text{Stop}_{x_i,a_i,x_j}$ **do**
3: **if** isConsistent(a_i, a_j) **then**
4: **if** searchPCwit^{rm}(a_i, a_j) **then**
5: $Last PC_{x_i,a_i,x_j} \leftarrow a_j$;
6: $Last AC_{x_i,a_i,x_j} \leftarrow a_j$;
7: return TRUE;
8: $a_j \leftarrow$ next value in $D(x_j)$;
9: return FALSE;

by the first value. Once a branching decision is made (e.g. variable assignment), `maxRPC3-resOpt` copies all the *Last PC* residues to the *Stop* data structure. Then, as `maxRPC3-resOpt` is executed at this specific node, the search for a new PC-support for $a_i \in D(x_i)$ in $D(x_j)$ starts from the value immediately after $Last PC_{x_i,a_i,x_j}$, continues through the end of the domain, if no PC-support is found, and back to the start of the domain until it encounters $Stop_{x_i,a_i,x_j}$. This may save many checks since, unlike `maxRPC3`^{rm}, each value in $D(x_j)$ can be checked for PC-support at most once.

We now explain in detail functions *checkPCsupLoss-resOpt* and *checkPCwitLoss-resOpt*, that replace functions *checkPCsupLoss*^{rm} and *checkPCwitLoss*^{rm}. In function *checkPCsupLoss-resOpt* (Function 10), we set a_j to the next value after $Last PC_{x_i,a_i,x_j}$, which is the first value to be checked for being a PC-support in line 1. When the search for PC-support encounters $Stop_{x_i,a_i,x_j}$ (line 2), all possible PC-supports will have been examined. Note that since we consider the domains to be circular, once the last available value in $D(x_j)$ has been unsuccessfuly checked, the search for PC-support will continue from the start of $D(x_j)$. That is, in line 8 a_j will be set to the first available value in $D(x_j)$.

A significant difference from `maxRPC3`^{rm} is that `maxRPC3-resOpt` cannot exploit the bidirectionality of *Last PC*. When a PC-support $a_j \in D(x_j)$ is found for $a_i \in D(x_i)$ then only $Last PC_{x_i,a_i,x_j}$ is set to a_j. We do not set $Last PC_{x_j,a_j,x_i}$ to a_i, as done in `maxRPC3`^{rm}, because bidirectionality no longer holds. To demonstrate this, assume that during the application of `maxRPC3-resOpt` at some node, we discover the PC-support $a_j \in D(x_j)$ for $a_i \in D(x_i)$ and through bidirectionality $Last PC_{x_j,a_j,x_i}$ is set to a_i. Now a later point in search when `maxRPC3`^{rm} is invoked we set $Stop_{x_j,a_j,x_i}$ to a_i and continue propagation. If during the search for PC-support for value $a'_i \in D(x_i)$, $a'_i \neq a_i$, in $D(x_j)$ we discover a_j then $Last PC_{x_j,a_j,x_i}$ will be set to a'_i. Now assume that later we seek a PC-support for a_j in $D(x_i)$ and a'_i is no longer valid. Then all values located between a_i and a'_i will be skipped because the search will start at a'_i+1 and will terminate when $a_i = Stop_{x_j,a_j,x_i}$ is reached. Consequently, bidirectionality cannot be exploited. To this end, the auxilary variable *rm*, used in *initialization* function is set to *false* to skip line 11.

On the other hand, *Last AC* is used as in `maxRPC3`^{rm} and thus it is updated when a PC-support is found, since this is also an AC-support. Furthermore, the search for a PC-witness for a pair of values is conducted by *searchPCwit*^{rm}, as the changes concern *Last PC* and do not affect *Last AC*.

Function *checkPCwitLoss-resOpt* is called when a_j is not removed by *checkPCsupLoss-resOpt*. The pseudocode is simply described in textual form, since it is the same as in *checkPCwitLoss*^{rm} (Function 9) until line 11. The second part of

the function (line 12) is executed when $LastPC_{x_i,a_i,x_k}$ is not valid (line 3), or because there is no PC-witness for the pair (a_i, a_k) in $D(x_j)$. In these cases a new PC-support for a_i is sought in $D(x_k)$, and this is done essentially in the same way as in function *checkPCsupLoss-resOpt* (Function 10).

Comparing with previous algorithms, `maxRPC3-resOpt` is sound and complete, as no supports nor witnesses can be overlooked and thus the proof of correctness is very similar to the one given for `maxRPC3`^{rm}. The node time complexity of `(l)maxRPC3-resOpt` is O(end^3), the same as `maxRPC3`, since the search for a new PC-support starts from $LastPC+1$ and not from scratch as in `maxRPC3`^{rm}. Before `maxRPC3-resOpt` is invoked, we set $Stop=LastPC$ and thus 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 nd times for PC-support while going down the branch. As a result the branch complexity is O(en^2d^4). The node space complexity is determined by the space required for storing the $LastAC$, $LastPC$, and $Stop$ structures, which is O(ed). The branch space complexity is also O(ed), because the data structures are not copied/restored.

Although `maxRPC3-resOpt` achieves a better node complexity than `maxRPC3`^{rm}, it carries the additional overhead of having to initialize the $Stop$ data structure at each node of the search tree. Experiments in Section 6 show that this is indeed an important drawback. The copying of $LastPC$ to $Stop$ at each node (in $O(ed)$ time) results in higher cpu times, despite the savings in constraint checks.

4.2 maxRPC3-start

A simple way to reduce the number of constraint checks, when a value in $LastPC$ is not valid, is to keep track of the first PC-support found after preprocessing. The version of `maxRPC3`^{rm} presented here, called `maxRPC3-start`, stores this value in a structure we call $LeftMostPC$, with $O(ed)$ size. In case of PC-support loss, instead of searching from scratch, we start from the value stored in $LeftMostPC$ that contains the first PC-support found in the *initialization* function. Thus, we omit values between the first value in a domain and the $LeftMostPC$ value to save redundant checks. For every value $a_i \in D(x_i)$ and constraint $c_{i,j}$, $LeftMostPC_{x_i,a_i,x_j}$ is initialized to NIL, like the $LastPC$ and $LastAC$ structures, and it is updated in the *initialization* function, exactly when $LastPC$ is updated. To obtain algorithm `maxRPC3-start` from `maxRPC3`^{rm}, we make the following simple changes.

– We insert in line 11 of *initialization* the assignment:

$$LeftMostPC_{x_i,a_i,x_j} \leftarrow a_j;$$

  Note that while we still exploit the bidirectionality of $LastPC$, this property does not hold for $LeftMostPC$. That is, if the first PC-support for a_i in $D(x_j)$ is value a_j, this does not necessarily mean that the first PC-support for a_j in $D(x_i)$ is a_i.
– In order to start the search for a new PC-support from the first PC-support found, we replace line 1 in *checkPCsupLoss*^{rm} with:

$$1: \quad \textbf{\textit{for each}}\ a_j \in D(x_j), a_j \geq LeftMostPC_{x_i,a_i,x_j}\ \textbf{\textit{do}}$$

  This change will affect also the *checkPCwitLoss*^{rm} function which calls *checkPCsupLoss*^{rm}in line 12.

Table 1 Time and space complexities of (l)maxRPC algorithms

Algorithm	Time complexity	Space complexity	Maintains structures
maxRPC1	$O(end^3)$	$O(end)$	Yes
maxRPC2	$O(end^3)$	$O(end)$	Yes
maxRPC3	$O(end^3)$	$O(end)$	Yes
maxRPC^{rm}	$O(en^2d^4)$	$O(end)$	No
maxRPC3^{rm}	$O(en^2d^4)$	$O(ed)$	No
maxRPC3-resOpt	$O(en^2d^4)$	$O(ed)$	No
maxRPC3-start	$O(en^2d^4)$	$O(ed)$	No
lmaxRPC2	$O(end^3)$	$O(end)$	Yes
lmaxRPC3	$O(end^3)$	$O(end)$	Yes
lmaxRPC^{rm}	$O(end^4)$	$O(ed)$	No
lmaxRPC3^{rm}	$O(end^4)$	$O(ed)$	No
lmaxRPC3-resOpt	$O(end^4)$	$O(ed)$	No
lmaxRPC3-start	$O(end^4)$	$O(ed)$	No

`maxRPC3-start` is sound and complete as it is guaranteed than no value earlier than the corresponding $LeftMostPC$ value can be a potential PC-support for some value $a_i \in D(x_i)$. The node and branch time complexity of `maxRPC3-start` is $O(en^2d^4)$, the same as `maxRPC3`^{rm}, as in the worst case, the $LeftMostPC$ values are the first values in each variable's domain. `lmaxRPC3-start` is the light version that results from removing the corresponding *checkPCwitLoss-start* function. Its complexity is $O(end^4)$, the same as `lmaxRPC3`^{rm}.

Table 1 summarises the asymptotic branch time and space complexities of the available (l)maxRPC algorithms. Under the column "maintains structures" we indicate whether a given algorithm requires to incrementally maintain some data structure or not.

5 Heuristics for maxRPC algorithms

Numerous heuristics for ordering constraint or variable revisions have been proposed and used within AC algorithms [2, 11, 17, 29]. Generally, many constraint solvers employ heuristics to order the application of propagators or/and the revision of variables and constraints [27]. 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. 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 in many ways in either a maxRPC or a lmaxRPC algorithm. In the following we summarize the possibilities of heuristic using algorithm `(l)maxRPC3` for illustration.

H1 A heuristic can be used to select the next variable x_j to remove from the propagation list Q (line 4 of Algorithm 1). Such heuristics are successfully used within AC algorithms.

H2 After a variable x_j is removed from Q 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 (line 5 of Algorithm 1).

H3 Once an AC-support $a_j \in D(x_j)$ has been found for a value $a_i \in D(x_i)$, we try 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 (line 1 of Function *searchPCwit*).

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

H4 For each value $a_i \in D(x_i)$ and each variable x_k constrained with both x_i and x_j, Function *checkPCwitLoss* checks if the pair (a_i, a_k) still has a PC-witness in $D(x_j)$. Again heuristics can be used to determine the order in which the variables constrained with x_i and x_j are considered (line 1 of *checkPCwitLoss*).

H5 In Function *checkPCwitLoss*, a new PC-support for a_i in $D(x_k)$ may be seeked. The order in which variables constrained with both x_i and x_k are considered can be determined heuristically as in the case of H3 above (within the call to *searchPCwit*).

As explained, the purpose of such ordering heuristics will be to "fail-first" [19]. That is, to quickly discover potential failures (in the case of H2 above), refute values that are not PC-supports (H3 and H5) and delete values that have no PC-support (H3). Such heuristics can be applied within any coarse-grained maxRPC algorithm to decide the order in which variables are considered. Examples of heuristics that can be used are the following.

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

wdeg For H1 consider the variables in descending weighted degree. For H2 consider the variables x_i in descending weight for the constraint $c_{i,j}$. In the case of H3 consider the variables x_k in descending average weight for the constraints $c_{i,k}$ and $c_{j,k}$. Similarly for H4 and H5.

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

Experiments demonstrated that applying heuristics H1 and H2 can sometimes be effective, while doing so for H3, H4, and H5 may save constraint checks but usually penalizes cpu times because of the overhead involved in computating the heuristics. 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 the case of H2 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 [2].

6 Experiments

To evaluate the various maxRPC algorithms, we 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 (see Table 2). More details about these instances can be found in C.Lecoutre's homepage.[4]

All algorithms used the dom/wdeg heuristic for variable ordering [12] and lexicographic value ordering. As explained in Section 2, dom/wdeg increases the weight of a constraint when this constraint causes a value removal. This process is rather straightforward when AC is used for constraint propagation, but perhaps not so when stronger local consistencies are used. For the case of maxRPC we chose to increase constraint weights in the following way. When a failure occurs, the weight of constraint $c_{i,j}$ is updated, right after line 7 and 11 of Algorithm 1 and after line 13 in the *initialization* function.

In all following tables, the results of the best algorithm, with respect to run-time, are highlighted with bold. If not explicitly mentioned, the propagation list Q was implemented as a FIFO queue and no heuristic from Section 5 was used.

Table 2 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 coarse-grained maxRPC algorithms, `maxRPC2`, `maxRPC3`, `maxRPC`^{rm}, `maxRPC3`^{rm} and from their corresponding light versions.

Regarding existing algorithms, results demonstrate that `maxRPC`^{rm} is particularly costly on large instances because of the penalties associated in initializing its data structures. Specifically, this algorithm timed out on some large instances of the Queens problem class, which explains the empty data entries in the table. In comparison, `maxRPC2` displays a better average performance which is not surprising given its lower complexity. The new algorithm `maxRPC3` is very close to `maxRPC2` in run times, apart from the first and last classes where it is notably faster. The same holds for `maxRPC3`^{rm} with the exception of the geometric class where it is clearly worse than the rest of the algorithms. Any gain in performance displayed by the new algorithms is due to the elimination of many redundant constraint checks as the corresponding numbers show.

Comparing light to full maxRPC algorithms it is perhaps surprising that the light versions typically achieve the same number of value deletions as their full counterparts. This means that approximation algorithms for maxRPC are quite effective. Any differences in value deletions among maxRPC algorithms are caused by the different order of operations in which inconsistency is discovered for some instances. In classes where the constraints checks for a maxRPC and a corresponding lmaxRPC algorithm are the same or very close, there are very few, if any, value deletions.

[4] http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

Table 2 Mean stand-alone performance in all 200 instances grouped by problem class

Problem class		maxRPC2	maxRPC3	maxRPC^{rm}	maxRPC3^{rm}	lmaxRPC2	lmaxRPC3	lmaxRPC^{rm}	lmaxRPC3^{rm}
RLFAP	t	1.581	1.125	5.754	1.064	0.942	**0.928**	0.929	0.931
(scen, graph)	rm	3,458	3,458	3,456	3,458	3,458	3,458	3,458	3,458
	cc	15.2M	8.9M	14.9M	8.2M	7.2M	7.1M	6.6M	7.2M
Random	t	0.149	0.153	**0.121**	0.146	0.148	0.151	0.156	0.149
(modelB, forced)	rm	20	20	21	21	25	25	25	31
	cc	0.181M	0.179M	0.181M	0.179M	0.178M	0.177M	0.178M	0.178M
Graph coloring	t	1.076	1.001	1.146	1.009	0.981	0.987	0.988	**0.980**
	rm	255	255	255	255	255	255	255	255
	cc	17M	16.1M	16.9M	16M	15.8M	15.8M	15.8M	15.8M
Quasigroup	t	0.211	0.201	0.276	0.215	0.173	**0.166**	0.173	0.174
(qcp, qwh bqwh)	rm	1,167	1,167	1,167	1,167	1,167	1,167	1,167	1,167
	cc	0.67M	0.43M	0.62M	0.42M	0.43M	0.38M	0.41M	0.38M
Geometric	t	0.217	0.214	**0.163**	0.336	0.222	0.213	0.214	0.218
	rm	0	0	0	0	0	0	0	0
	cc	0.33M	0.33M	0.33M	0.33M	0.33M	0.33M	0.33M	0.33M
QueensKnights	t	30.705	29.724	–	29.310	27.827	28.073	27.791	**27.732**
Queens,	rm	96	96	–	96	96	96	96	96
QueenAttack	cc	426M	390M	–	389M	366M	366M	366M	366M
Driver, haystacks	t	1.449	1.107	1.781	1.086	0.996	**0.931**	0.979	1.002
blackHole	rm	247	247	247	247	247	247	247	247
job-shop	cc	14.4M	10M	13.5M	9.9M	9.3M	8.9M	9.3M	8.9M

Cpu times (t) in secs, removed values (rm) and constraint checks (cc) are given

Table 3 compares the performance of search algorithms that apply lmaxRPC throughout search on several problem classes including instances from RLFAPs, random, Quasigroup, geometric, and Queen problems. These instances have been selected to demonstrate cases where either the new algorithms achieve a clear improvement making the best algorithm among them outperform or compete with MAC, or cases where, despite the improvement, the maxRPC-based algorithms are still significantly inferior to MAC. Hence, we present some extreme behaviors for both situations. The algorithms compared are `lmaxRPC`^{rm}, `lmaxRPC3`^{rm}, `lmaxRPC3-resOpt` and `lmaxRPC3-start`. We do not present results from

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

Instance		AC3^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC3-resOpt	lmaxRPC3-start
scen11-f7	t	**109.3**	482.6	186.6	214	159.4
	n	353,901	76,954	76,954	57,037	76,954
	cc	467M	5,184M	1,596M	1,011M	1,323M
graph9-f9	t	**8.7**	54.5	21	45.6	17.5
	n	46,705	16,839	16,839	14,838	16,839
	cc	25M	458M	184M	145M	153M
rand-2-40-11-414-200-30	t	14.1	17.6	11.6	**8.9**	11.5
	n	164,958	28,655	28,655	21,105	28,655
	cc	61M	249M	98M	70M	97M
will199GPIA-6	t	**1.6**	5.9	2.5	7.9	2.7
	n	6,996	3,316	3,316	4,300	3,316
	cc	6M	53M	21M	22M	20M
qcp150-120-5	t	22.9	29.3	**15.5**	73.8	15.7
	n	525,629	130,384	130,384	237,644	130,384
	cc	37M	265M	43M	69M	42M
qcp150-120-9	t	95.2	120.6	**57.7**	157.4	59
	n	2,437,173	627,679	627,679	617,662	627,679
	cc	157M	1,060M	163M	151M	162M
qwh20-166-1	t	15.3	21.7	**12.2**	42.7	12.8
	n	234,095	54,286	54,286	31,346	54,286
	cc	19M	156M	18M	10M	18M
qwh20-166-6	t	758.3	462.5	**245.9**	3,342.5	256.2
	n	10,691,633	984,555	984,555	2,364,104	984,555
	cc	911M	3,381M	377M	921M	372M
qwh20-166-7	t	64.5	46.2	**24.7**	319.3	26.3
	n	1,050,144	124,212	124,212	241,184	124,212
	cc	85M	342M	40M	75M	39M
geo50-20-d4-75-1	t	**54.4**	248.2	140.5	143.7	145.7
	n	260,996	122,750	122,750	124,535	122,750
	cc	6M	1,454M	377M	1,376M	375M
queenAttacking6	t	32.9	60.8	**23.9**	94	24.3
	n	234,759	18,488	18,488	137,731	18,488
	cc	104M	888M	242M	860M	238M
queensKnights-15-5-mul	t	**3.1**	27.6	16.5	13.4	11.6
	n	5,819	3,5862	3,586	2,924	3,586
	cc	23M	462M	233M	174M	183M
haystacks-05	t	4.5	2.6	2	2.8	**1.8**
	n	1,182,023	167,629	167,629	223,547	167,629
	cc	13M	13M	7M	10M	6M

maxRPC^{rm} and maxRPC3^{rm}, since these two algorithms, and especially maxRPC^{rm}, are inferior to the light versions when used during search. To be precise, maxRPC3^{rm} is competitive on some instances but clearly worse on average. On the other hand, maxRPC^{rm} is substantially slower on all the tested instances and exceeds the time limit of two hours on the hardest among them. Algorithms (l)maxRPC2 and (l)maxRPC3 are even less competitive when used during search, because of the overheads for the copying and restoration of the *Last PC* and *Last AC* data structures. (l)maxRPC3 is typically more efficient than (l)maxRPC2.

In general, any maxRPC algorithm is clearly inferior to the corresponding light version when applied during search. The reduction in visited nodes achieved by the former is relatively small and does not compensate for the higher run times of enforcing maxRPC. To put the performance of the lmaxRPC algorithms in perspective, we include results from MAC3^{rm} which is considered one of the most efficient versions of MAC [22, 23]. All of the algorithms used a 2-way branching scheme.[5]

Experiments showed that lmaxRPC^{rm} is the most efficient among existing algorithms when applied during search, which confirms the results given in [28]. Accordingly, lmaxRPC3^{rm} is the most efficient among our algorithms. It is over two times faster than lmaxRPC^{rm} on hard instances, while algorithms lmaxRPC3-resOpt and lmaxRPC3-start are also competitive in many instances. The overhead of copying *Last PC* to *Stop* causes lmaxRPC3-resOpt to slow down search in many cases, despite the reduction in the number of constraint checks.

Instance qwh20-166-6 is a pathological case for lmaxRPC3-resOpt as this algorithm requires considerable effort compared to the other algorithms. Recall that this algorithm does not exploit the bidirectionality of support, as explained in Section 4.1, meaning that variable revisions, constraint checks, and failures may occur in different orders compared to other algorithms. Through the interaction with the dom/wdeg variable ordering heuristic this may cause a different search direction (see discussion at the end of Section 5), explaining the pathological case.

lmaxRPC3-start and lmaxRPC3^{rm} have similar performance when the numbers of constraint checks are similar. More precisely, lmaxRPC3-start is better only when the PC-support found in preprocessing is lexicographically bigger from the first value in any domain. Since this case does not occur very often, there are no significant benefits when compared to lmaxRPC3^{rm} that starts searching from scratch.

Importantly, the speed-ups obtained can make a search algorithm that efficiently applies lmaxRPC competitive with MAC on many instances. For instance, in qwh20-166-6 lmaxRPC3^{rm} achieves a better run time than MAC by a factor of three while lmaxRPC^{rm} is 2 times slower compared to lmaxRPC3^{rm}.

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 2 times can be obtained (e.g. qcp150-120-9 and qwh20-166-6). However, there are still many instances where MAC remains considerably faster despite the improvements (e.g. graph9-f9, geo50-20-d4-75-1).

[5]The results reported in [1] were obtained using a *d*-way branching scheme.

Table 4 Mean search performance in all 200 instances grouped by class

Problem class		AC3^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC3-resOpt	lmaxRPC3-start
RLFAP	t	**13.5**	61.7	21.7	26.7	18.8
(scen, graph)	n	42,250	8,727	8,727	7,326	8,727
	cc	56M	625M	201M	139M	166M
Random	t	**2.7**	5.8	3.6	3.9	3.6
(modelB, forced)	n	29,538	7,385	7,385	7,835	7,385
	cc	11M	82M	30M	31M	30M
Graph coloring	t	**4.8**	61	25	45.9	29.7
	n	3,910	2,225	2,225	2,866	2,225
	cc	12M	984M	284M	303M	283M
Quasigroup	t	55.6	47.8	**22**	231.7	22.8
(qcp, qwh, bqwh)	n	866,099	117,974	117,974	204,407	117,974
	cc	70M	315M	39M	71M	38M
Geometric	t	**11.3**	50.6	29.1	30.3	30.1
	n	55,825	26,687	26,687	27,656	26,687
	cc	55M	721M	314M	314M	313M
QueensKnights,	t	**7.1**	133.6	42.7	56.3	42.5
Queens,	n	38,663	4,829	4,829	22,321	4,829
QueenAttack	cc	27M	1,583M	563M	655M	552M
Driver, blackHole,	t	**1.6**	14.7	5.6	15.5	5.7
haystacks,	n	115,717	28,750	28,750	34,001	28,750
job-shop	cc	3M	141M	33M	35M	32M

In Table 4 we summarize the results of our experiments by giving averages over different problem classes. These results demonstrate that `lmaxRPC3`^{rm} outperforms `lmaxRPC`^{rm} in all problem classes, often considerably. This was the case in all 200 instances tried. Algorithms `lmaxRPC3-resOpt` and especially `lmaxRPC3-start` display similar performance to `lmaxRPC3`^{rm}. `lmaxRPC3-resOpt` displays its worst performance in quasigroup problems where it performs twice as much constraint checks on average. Taking also into account that `lmaxRPC3-resOpt` copies *Last PC* to *Stop* explains the variance in the results given in Tables 3 and 4. In general, `lmaxRPC3`^{rm} is competitive with MAC on RLFAP and random instances and outperforms it on the Quasigroup classes. In contrast, `lmaxRPC3`^{rm} is clearly inferior to `AC3`^{rm} on Queens class and in the last category that includes instances from various other structured problem classes.

6.1 *d*-way branching

We have also experimented with the above search algorithms under the *d*-way branching scheme using again the dom/wdeg heuristic for variable ordering. Table 5 reports results from the same instances as Table 3, in order to directly compare our algorithms on the two different branching schemes. We exclude `lmaxRPC3-resOpt` which is the less competitive among the algorithms of Table 3. We can observe that `lmaxRPC3`^{rm} is faster by a factor of two on the RLFAP instance graph9-f9, while with 2-way branching `AC3`^{rm} was superior. Differences in the relative performance of AC and maxRPC occur in other problems as well (e.g. random, quasigroup and queensAttacking). For example, in qwh instances `lmaxRPC3`^{rm} has better run-time results against `AC3`^{rm} but not by as large margins as under 2-way

Table 5 Cpu times (t) in secs, nodes (n) and constraint checks (cc) from various problem instances when d-way branching is used

Instance		AC3^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC3-start
scen11-f7	t	981.9	3,338	1,128	**870**
	n	3,696,154	552,907	552,907	552,907
	cc	4,287M	31,098M	9,675M	8,193M
graph9-f9	t	57.1	85.3	33.3	**30.9**
	n	273,766	26,276	26,276	26,276
	cc	158M	729M	290M	242M
rand-2-40-11-414-200-30	t	**11.3**	33.4	26.9	21.5
	n	110,091	49,100	49,100	49,100
	cc	51M	484M	189M	187M
will199GPIA-6	t	**3**	11.7	4.8	5.1
	n	13,243	4,971	4,971	4,971
	cc	13M	108M	42M	41M
qcp150-120-5	t	**15.9**	34.8	17.8	18.6
	n	233,311	100,781	100,781	100,781
	cc	27M	330M	54M	53M
qcp150-120-9	t	**66.5**	162.9	78.4	81.2
	n	2,437,173	627,679	627,679	627,679
	cc	157M	1,060M	163M	162M
qwh20-166-1	t	**14.6**	28.2	15.7	16.4
	n	234,095	54,286	54,286	54,286
	cc	19M	156M	18M	18M
qwh20-166-6	t	462.7	674.3	**346.2**	367
	n	4,651,632	919,861	919,861	919,861
	cc	633M	5,089M	566M	558M
qwh20-166-7	t	30	51.9	**27.8**	29.1
	n	263,713	76,624	76,624	76,624
	cc	42M	392M	45M	44M
geo50-20-d4-75-1	t	**38.7**	144.8	81.9	82.3
	n	181,560	79,691	79,691	79,691
	cc	192M	2,045M	880M	876M
queenAttacking6	t	**54.7**	406.8	153.9	146.6
	n	262,087	103,058	103,058	103,058
	cc	211M	6,035M	1,640M	1,623M
queensKnights-15-5-mul	t	**18.2**	82.6	40.4	23.5
	n	35,445	13,462	13,462	13,462
	cc	154M	963M	387M	282M
haystacks-05	t	0.7	0.7	0.8	0.7
	n	110,638	20,278	20,278	20,278
	cc	1.4M	1.9M	1.1M	1.0M

branching. In comparison to `lmaxRPC`^{rm}, `lmaxRPC3`^{rm} remains advantageous in all instances.

Table 6 summarizes results from the application of lmaxRPC during search using d-way branching. We give average results for all the tested instances, grouped into specific problem classes, as in Table 4. As can be seen, `lmaxRPC3`^{rm} and `lmaxRPC3-start` improve on the existing best algorithm considerably, making lmaxRPC outperform MAC on the quasigroup problem classes and be quite competitive on the RLFAP class. As expected, when comparing the same (AC or maxRPC)

Table 6 Mean search performance in all 200 instances grouped by class, when d-way branching is used

Problem class		AC3^{rm}	lmaxRPC^{rm}	lmaxRPC3^{rm}	lmaxRPC3-start
RLFAP	t	124	157.3	157.3	**123.2**
(scen, graph)	n	424,128	74,083	73,083	73,083
	cc	559M	4,394M	1,387M	1,092M
Random	t	**2.2**	7.7	5.3	4.8
(modelB, forced)	n	19,809	9,270	9,270	9,270
	cc	9M	110M	40M	40M
Graph coloring	t	**8.9**	110.3	46.9	43.4
	n	5,919	2,983	2,983	2,983
	cc	23M	1,735M	455M	454M
Quasigroup	t	35.5	57.4	**29.6**	31.2
(qcp,qwh,bqwh)	n	387,495	103,994	103,994	103,994
	cc	51M	458M	56M	55M
Geometric	t	**8.2**	29.7	17.2	17.3
	n	39,879	17,273	17,273	17.273
	cc	39M	418M	180M	179M
QueensKnights,	t	**14.7**	206.1	73.6	67.6
Queens,	n	67,019	24,859	24,859	24,859
QueenAttack	cc	73M	2,796M	839M	807M
Driver,blackHole,	t	**0.8**	13.2	5.2	5.3
haystacks,	n	13,075	11,349	11,349	11,349
job-shop	cc	1M	121M	25M	25M

algorithm under the two different branching schemes, 2-way branching is typically superior.

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 some classes of binary CSPs with either of the two standard branching schemes. This confirms the conjecture of [15] about the potential of maxRPC as an alternative to AC. In addition, our results, along with ones in [28], show that approximating strong and complex local consistencies can be very beneficial.

6.2 Heuristics

We have also run experiments to evaluate several of the heuristics described in Section 5. In these experiments we have mainly used the best algorithm, `lmaxRPC3`^{rm}, under 2-way braching. Intuitively, the use of heuristics may improve the algorithm's performance as explained in Section 5. Since only light versions of maxRPC are practical for use during search, we have only tested heuristics H1, H2 and H3. Recall that heuristics H4 and H5 are not applicable for light maxRPC algorithms.

With respect to the specific strategy for ordering variables under the different heuristics, we have tried all the "fail-first" methods analyzed in Section 5 (i.e. dom, del_ratio, wdeg, dom/wdeg). dom and wdeg were not as efficient as the other methods and are thus ommitted from Table 7. The algorithm used is `lmaxRPC3`^{rm}, except from the last column where we report results from `lmaxRPC3-start`. Apart

Table 7 Mean search performance in all 200 instances grouped by class, when different heuristics are used

Problem class		AC3^{rm}	lmaxRPC3^{rm}	H1	H2	H1+H2	H1+H2+H3	H1+H2(*del_ratio*)	lmaxRPC3-start+H1+H2
RLFAP	t	**13.5**	21.7	18.8	18.5	18	22.6	22.5	**13.9**
(scen, graph)	n	42,250	8,727	7,993	9,059	7,940	7,940	8,624	7,940
	cc	56M	201M	157M	143M	152M	163M	200M	130M
Random	t	**2.7**	3.6	3.3	7.1	5.4	9.7	5.1	5.4
(modelB, forced)	n	29,538	7,385	7,508	15,431	11,452	11,452	10,369	11,452
	cc	11M	30M	25M	55M	38M	37M	39M	38M
Graph coloring	t	**4.8**	25	24.7	25.9	28.5	259.9	25	27.7
	n	3,910	2,225	2,163	2,175	2,043	2,043	2,079	2,043
	cc	12M	284M	248M	254M	236M	236M	218M	258M
Quasigroup	t	55.6	22	**20.6**	37.2	34.9	101.1	26.8	35.9
(qcp, qwh, bqwh)	n	866,099	117,974	109,945	159,440	143,206	143,206	122,291	143,206
	cc	70M	39M	35M	54M	47M	47M	38M	46M
Geometric	t	**11.3**	29.1	21.8	18.2	17.5	30.9	31.1	17.5
	n	55,825	26,687	24,938	17,184	18,295	18,295	26,580	18,295
	cc	55M	314M	224M	178M	164M	159M	310M	164M
QueensKnights,	t	**7.1**	42.7	50.7	46.8	53.1	154.3	44.1	51
Queens,	n	38,663	4,829	13,452	8,469	10,034	10,034	4,849	10,034
QueenAttack	cc	27M	563M	624M	588M	649M	640M	563M	616M
Driver, blackHole,	t	**1.6**	5.6	6.7	7	6.4	27.3	7.3	5.9
haystacks,	n	115,717	28,750	51,511	53,685	56,148	56,148	29,761	56,148
job-shop	cc	3M	33M	30M	31M	31M	32M	33M	31M

from column $H1 + H2(del\_ratio)$, where the heuristic is mentioned explicitly, in the rest of the columns we use dom/wdeg.

Considering the results in Table 7 compared to results in Table 4 it seems that the application of heuristics does not offer any benefits as the algorithm's performance is marginally improved, if at all. In some problem classes using no heuristic at all is the best choice. These results, obtained using 2-way branching, are in contrast to results from [1] where it was shown that heuristics H1 and H2 are mildly beneficial when d-way branching is used.

The most promising is the application of both H1 and H2 (H1+H2), where x_j extracted from Q and x_i, which is the neighbouring variable of x_j, are ordered in ascending order of the dom/wdeg value. The less efficient combination is the H1+H2+H3 because of the run-time overhead caused by the often computation of all three heuristics. Comparing del_ratio and dom/wdeg on H1+H2 we conclude that the former is preferable on Quasigroup and Queen problems while the latter is better on RLFAP and Geometric problems. On the rest of the problem classes they display similar performance.

6.3 Interleaving AC and maxRPC

Since there are problem classes where either an algorithm that maintains AC or one that maintains lmaxRPC is preferable, we have experimented with hybrid propagation schemes that interleave `lmaxRPC3`^{rm} and `AC3`^{rm}. Specifically, we have considered the following simple ways to interleave the two algorithms under 2-way

Table 8 Mean hybrid search performance in all 200 instances grouped by class

Problem class		AC3^{rm}	lmaxRPC3^{rm}	$(x = a) \bigwedge (x \neq a)$	
				lmaxRPC3$^{rm} \bigwedge$ AC3^{rm}	AC3$^{rm} \bigwedge$ lmaxRPC3^{rm}
RLFAP	t	**13.5**	21.7	21.9	19.5
(scen,graph)	n	42,250	8,727	17,231	25,748
	cc	56M	201M	193M	103M
Random	t	**2.7**	3.6	3.2	4.1
(modelB,forced)	n	29,538	7,385	11,488	16,668
	cc	11M	30M	24M	30M
Graph coloring	t	**4.8**	25	21.6	5.9
	n	3,910	2,225	2,745	2,654
	cc	12M	284M	240M	58M
Quasigroup	t	55.6	**22**	30.2	40.9
(qcp,qwh,bqwh)	n	866,099	117,974	233,919	324,373
	cc	70M	39M	43M	60M
Geometric	t	**11.3**	29.1	16.6	15.3
	n	55,825	26,687	25,042	28,785
	cc	55M	314M	164M	140M
QueensKnights,	t	**7.1**	42.7	42.2	9.9
Queens,	n	38,663	4,829	9,645	11,648
QueenAttack	cc	27M	563M	535M	211M
Driver, blackHole	t	**1.6**	5.6	2.1	**1.8**
haystacks, job-shop	n	115,717	28,750	64,891	86,446
	cc	3M	33M	30M	31M

branching: At any left branch we run lmaxRPC3^{rm} (respectively AC3^{rm}) after a value assigment, while at any right branch we run AC3^{rm} (respectively lmaxRPC3^{rm}) after a value removal. Table 8 summarizes the results of our experiments with these methods.

Given the results in Table 8, the first observation we can make is that none of the two hybrid propagation schemes is substantially worse than both lmaxRPC3^{rm} and AC3^{rm} on any problem class. In contrast, there are problem classes where the hybrids outperform either maxRPC (e.g. geometric) or AC (quasigroups) by substantial margins. This means that, as expected, the hybrid methods achieve a compromise between maxRPC and AC, which is evident by looking at both cpu times and node visits. Applying maxRPC at left branches results in performance closer to maintaining maxRPC, while when AC is applied at left branches the performance is closer to MAC. This is not surprising since the effects of constraint propagation are stronger after variable assignments compared to value removals. Therefore, the local consistency applied at left branches is the "dominant" one that determines the behaviour of the algorithm. As a result, the former hybrid method is better on quasigroup problems but worse on graph coloring and queens instances, while the two are close on the rest of the problem classes.

The preliminary results presented here give a strong indication that interleaving AC and stronger local consistencies, such as maxRPC, during search can be quite beneficial. Further research is certainly required to develop more informed and efficient ways of interleaving different local consistencies.

7 Conclusion

Although maxRPC has been identified as a promising strong local consistency for binary constraints, it has received rather narrow attention since it was introduced. Only two new algortihms have been proposed since the introduction of maxRPC1, the first algorithm for maxRPC, and they have only been evaluated on random problems, if at all.

In this paper we have identified sources of redundancies in the existing maxRPC algorithms which largely contribute to the high cost of maintaining maxRPC during search. Based on this, we presented 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 some problems, justifying the conjecture of [15] about the potential of maxRPC as an alternative to AC.

In the future it would be interesting to investigate the applicability of similar methods to efficiently achieve or approximate other local consistencies related to maxRPC such as PIC or maxRPCEn. Also, a very interesting direction is the efficient interleaved application of stronger consistencies, like maxRPC, and weaker but

cheaper ones, like AC. We have presented some initial results towards this, but further research is certainly required.

Acknowledgements We would like to thank the anonymous reviewers for their insightful comments that helped improve this paper.

References

1. Balafoutis, T., Paparrizou, A., Stergiou, K., & Walsh, T. (2010). Improving the performance of maxRPC. In *Proceedings of CP-2010* (pp. 69–83).
2. Balafoutis, T., & Stergiou, K. (2008). Exploiting constraint weights for revision ordering in arc consistency algorithms. In *ECAI-08 workshop on modeling and solving problems with constraints*.
3. Bartak, R., & Erben, R. (2004). A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS conference-2004*.
4. Berlandier, P. (1995). Improving domain filtering using restricted path consistency. In *Proceedings of IEEE CAIA-95* (pp. 32–37).
5. Bessiere, C. (1994). Arc-consistency and arc-consistency again. *Artificial Intelligence, 65*, 179–190.
6. Bessiere, C., Cardon, S., Debruyne, R., & Lecoutre, C. (2011). Efficient algorithms for singleton arc consistency. *Constraints, 16*, 25–53.
7. Bessiere, C., & Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI-2005* (pp. 54–59).
8. Bessière, C., Freuder, E. C., & Régin, J. C. (1995). Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95* (pp. 592–599).
9. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C. G., & Walsh, T. (2009). Decompositions of all different, global cardinality and related constraints. In *Proceedings of IJCAI-2009* (pp. 419–424).
10. Bessière, C., Régin, J. C., Yap, R., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence, 165*(2), 165–185.
11. Boussemart, F., Hemery, F., & Lecoutre, C. (2004). Revision ordering heuristics for the constraint satisfaction problem. In *CP-2004 workshop on constraint propagation and implementation, Toronto, Canada*.
12. Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of ECAI-2004* (pp. 482–486). Valencia, Spain.
13. Debruyne, R. (1999). A strong local consistency for constraint satisfaction. In *Proceedings of ICTAI-99* (pp. 202–209).
14. Debruyne, R., & Bessière, C. (1997). From restricted path consistency to max-restricted path consistency. In *Proceedings of CP-97* (pp. 312–326).
15. Debruyne, R., & Bessière, C. (2001). Domain filtering consistencies. *Journal of Artificial Intelligence Research, 14*, 205–230.
16. Freuder, E., & Elfe, C. (1996). Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI'96* (pp. 202–208).
17. Gent, I. P., MacIntyre, E., Prosser, P., Shaw, P., & Walsh, T. (1997). The constraindedness of arc consistency. In *Proceedings of CP-97* (pp. 327–340).
18. Grandoni, F., & Italiano, G. (2003). Improved algorithms for max-restricted path consistency. In *Proceedings of CP'03* (pp. 858–862).
19. Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*, 263–314.
20. Lecoutre, C., & Cardon, S. (2005). A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI-2005* (pp. 199–204).
21. Lecoutre, C., Cardon, S., & Vion, J. (2007). Conservative dual consistency. In *Proceedings of AAAI-07* (pp. 237–242).
22. Lecoutre, C., & Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of IJCAI-2007* (pp. 125–130).
23. Likitvivatanavong, C., Zhang, Y., Bowen, J., Shannon, S., & Freuder, E. (2007). Arc consistency during search. In *Proceedings of IJCAI-2007* (pp. 137–142).

24. Montanari, U. (1974). Network of constraints: Fundamental properties and applications to picture processing. *Information Science, 7*, 95–132.
25. Quimper, C. G., & Walsh, T. (2006). Global grammar constraints. In *Proceedings of CP-2006* (pp. 751–755).
26. Sabin, D., & Freuder, E. C. (1997). Understanding and improving the MAC algorithm. In *Proceedings of CP-1997* (pp. 167–181).
27. Schulte, C., & Stuckey, P. J. (2008). Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems, 31*(1), 1–43.
28. Vion, J., & Debruyne, R. (2009). Light algorithms for maintaining max-RPC during search. In *Proceedings of SARA-2009*.
29. Wallace, R., & Freuder, E. (1992). Ordering heuristics for arc consistency algorithms. In *AI/GI/VI* (pp. 163–169). Vancouver, British Columbia, Canada.