# Restricted Path Consistency Revisited

Kostas Stergiou(✉)

Department of Informatics and Telecommunications Engineering,
University of Western Macedonia, Kozani, Greece
kstergiou@uowm.gr

**Abstract.** Restricted path consistency (RPC) is a strong local consistency for binary constraints that was proposed 20 years ago and was identified as a promising alternative to arc consistency (AC) in an early experimental study of local consistencies for binary constraints. However, and in contrast to other strong local consistencies such as SAC and maxRPC, it has been neglected since then. In this paper we revisit RPC. First, we propose RPC3, a new lightweight RPC algorithm that is very easy to implement and can be efficiently applied throughout search. Then we perform a wide experimental study of RPC3 and a light version that achieves an approximation of RPC, comparing them to state-of-the-art AC and maxRPC algorithms. Experimental results clearly show that restricted RPC is by far more efficient than both AC and maxRPC when applied throughout search. These results strongly suggest that it is time to reconsider the established perception that MAC is the best general purpose method for solving binary CSPs.

## 1 Introduction

Restricted path consistency (RPC) is a local consistency for binary constraints that is stronger than arc consistency (AC). RPC was introduced by Berlandier [4] and was further studied by Debruyne and Bessiere [7,8]. An RPC algorithm removes all arc inconsistent values from a domain $D(x)$, and in addition, for any pair of values $(a, b)$, with $a \in D(x)$ and $b \in D(y)$ s.t. $b$ is the only support for $a$ in a $D(y)$, it checks if $(a, b)$ is path consistent. If it is not then $a$ is removed from $D(x)$. In this way some of the benefits of path consistency are retained while avoiding its high cost.

Although RPC was identified as a promising alternative to AC as far back as 2001 [8], it has been neglected by the CP community since then. In contrast, stronger local consistencies such as max restricted path consistency (maxRPC) [7] and singleton arc consistency (SAC) [8] have received considerable attention in the past decade or so [1–3,5,9,11,13,14]. However, despite the algorithmic developments on maxRPC and SAC, none of the two outperforms AC when maintained during search, except for specific classes of problems. Therefore, MAC remains the predominant generic algorithm for solving binary CSPs.

In this paper we revisit RPC and make two contributions compared to previous works that bring the state-of-the-art regarding RPC up to date. The first is algorithmic and the second experimental.


G. Pesant (Ed.): CP 2015, LNCS 9255, pp. 419–428, 2015.
DOI: 10.1007/978-3-319-23219-5_30

The two algorithms that have been proposed for RPC, called RPC1 [4] and RPC2 [7], are based on the AC algorithms AC4 and AC6 respectively. As a result they suffer from the same drawbacks as their AC counterparts. Namely, they use heavy data structures that are too expensive to maintain during search. In recent years it has been shown that in the case of AC lighter algorithms which sacrifice optimality display a better performance when used inside MAC compared to optimal but heavier algorithms such as AC4, AC6, AC7, and AC2001/3.1. Hence, the development of the residue-based version of AC3 known as AC3$^r$ [10,12]. A similar observation has been made with respect to maxRPC [1]. Also, it has been noted that cheap approximations of local consistencies such as maxRPC and SAC are more cost-effective than the full versions. In the case of maxRPC, the residue-based algorithm lmaxRPC3$^r$, which achieves an approximation of maxRPC, is the best choice when applying maxRPC [1].

Following these trends, we propose RPC3, an RPC algorithm that makes use of residues in the spirit of AC$^r$ and lmaxRPC$^r$ and is very easy to implement. As we will explain, for each constraint $(x, y)$ and each value $a \in D(x)$, RPC3 stores two residues that correspond to the two most recently discovered supports for $a$ in $D(y)$. This enables the algorithm to avoid many redundant constraint checks. We also consider a restricted version of the algorithm (simply called rRPC3) that achieves a local consistency property weaker than RPC, but still stronger than AC, and is considerably faster in practice.

Our second and most important contribution concerns experiments. Given that the few works on RPC date from the 90s, the experimental evaluations of the proposed algorithms were carried out on limited sets of, mainly random, problems. Equally importantly, there was no evaluation of the algorithms when used during search to maintain RPC. We carry out a wide evaluation on benchmark problems from numerous classes that have been used in CSP solver competitions. Surprisingly, results demonstrate that an algorithm that applies rRPC3 throughout search is not only competitive with MAC, but it clearly outperforms it on the overwhelming majority of tested instances, especially on structured problems. Also, it clearly outperforms lmaxRPC3$^r$. This is because RPC, and especially its restricted version, achieves a very good balance between the pruning power of maxRPC and the low cost of AC.

Our experimental results provide strong evidence of a local consistency that is clearly preferable to AC when maintained during search. Hence, perhaps it is time to reconsider the common perception that MAC is the best general purpose solver for binary problems.

## 2 Background

A *Constraint Satisfaction Problem* (CSP) is defined as a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of $n$ variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, with maximum cardinality $d$, and $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of $e$ constraints. In this paper we are concerned with binary CSPs. A binary constraint $c_{ij}$ involves variables $x_i$ and $x_j$.

At any time during the solving process if a value $a_i$ has not been removed from the domain $D(x_i)$, we say that the value is *valid*. A value $a_i \in D(x_i)$ is *arc consistent* (AC) iff for every constraint $c_{ij}$ there exists a value $a_j \in D(x_j)$ s.t. the pair of values $(a_i, a_j)$ satisfies $c_{ij}$. In this case $a_j$ is called an *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.

A pair of values $(a_i, a_j)$, with $a_i \in D(x_i)$ and $a_j \in D(x_j)$, is *path consistent* PC iff for any third variable $x_k$ constrained with $x_i$ and $x_j$ there exists a value $a_k \in D(x_k)$ s.t. $a_k$ is a support of both $a_i$ and $a_j$. In this case $a_j$ is a *PC-support* of $a_i$ in $D(x_j)$ and $a_k$ is a *PC-witness* for the pair $(a_i, a_j)$ in $D(x_k)$.

A value $a_i \in D(x_i)$ is *restricted path consistent* (RPC) iff it is AC and for each constraint $c_{ij}$ s.t. $a_i$ has a single support $a_j \in D(x_j)$, the pair of values $(a_i, a_j)$ is *path consistent* (PC) [4]. A value $a_i \in D(x_i)$ is *max restricted path consistent* (maxRPC) iff it is AC and for each constraint $c_{ij}$ there exists a support $a_j$ for $a_i$ in $D(x_j)$ s.t. the pair of values $(a_i, a_j)$ is *path consistent* (PC) [7]. A variable is RPC (resp. maxRPC) iff all its values are RPC (resp. maxRPC). A problem is RPC (resp. maxRPC) iff there is no empty domain and all variables are RPC (resp. maxRPC).

## 3 The RPC3 Algorithm

The RPC3 algorithm is based on the idea of seeking two supports for a value, which was first introduced in RPC2 [7]. But in contrast to RPC2 which is based on AC6, it follows an AC3-like structure, resulting in lighter use of data structures, albeit with a loss of optimality. As explained below, we can easily obtain a restricted but more efficient version of the algorithm that only approximates the RPC property. Crucially, the lack of heavy data structures allows for the use of the new algorithms during search without having to perform expensive restorations of data structures after failures.

In the spirit of AC$^r$, RPC3 utilizes two data structures, $\mathrm{R}^1$ and $\mathrm{R}^2$, which hold residual data used to avoid redundant operations. Specifically, for each constraint $c_{ij}$ and each value $a_i \in D(x_i)$, $\mathrm{R}^1_{x_i,a_i,x_j}$ and $\mathrm{R}^2_{x_i,a_i,x_j}$ hold the two most recently discovered supports of $a_i$ in $D(x_j)$. Initially, all residues are set to a special value NIL, considered to precede all values in any domain.

The pseudocode of RPC3 is given in Algorithm 1 and Function 2. Being coarse-grained like AC3, Algorithm 1 uses a propagation list $Q$, typically implemented as a fifo queue. We use a constraint-oriented description, meaning that $Q$ handles pairs of variables involved in constraints. A variable-based one requires minor modifications.

Once a pair of variables $(x_i, x_j)$ is removed from $Q$, the algorithm iterates over $D(x_i)$ and for each value $a_i$ first checks the residues $\mathrm{R}^1_{x_i,a_i,x_j}$ and $\mathrm{R}^2_{x_i,a_i,x_j}$ (line 5). If both are valid then $a_i$ has at least two supports in $D(x_j)$. Hence, the algorithm moves to process the next value in $D(x_i)$. Otherwise, function *findTwoSupports* is called. This function will try to find two supports for $a_i$ in $D(x_j)$. In case it finds none then $a_i$ is not AC and will thus be deleted (line 13).

**Algorithm 1.** RPC3:**boolean**

1: **while** Q $\neq \emptyset$ **do**
2: Q $\leftarrow$ Q$-\{(x_i, x_j)\}$;
3: Deletion $\leftarrow$ FALSE;
4: **for each** $a_i \in D(x_i)$ **do**
5: **if** both $R^1_{x_i,a_i,x_j}$ and $R^2_{x_i,a_i,x_j}$ are valid **then**
6: **continue**;
7: **else**
8: **if** only one of $R^1_{x_i,a_i,x_j}$ and $R^2_{x_i,a_i,x_j}$ is valid **then**
9: R $\leftarrow$ the valid residue;
10: **else**
11: R $\leftarrow$ NIL;
12: **if** *findTwoSupports*$(x_i, a_i, x_j, R)$ = FALSE **then**
13: remove $a_i$ from $D(x_i)$;
14: Deletion $\leftarrow$ TRUE;
15: **if** D$(x_i) = \emptyset$ **then**
16: **return** FALSE;
17: **if** Deletion = TRUE **then**
18: **for each** $(x_k, x_i) \in C$ s.t. $(x_k, x_i) \notin Q$ **do**
19: Q $\leftarrow$ Q $\cup \{(x_k, x_i)\}$;
20: **for each** $(x_l, x_k) \in C$ s.t. $x_l \neq x_i$ **and** $(x_l, x_i) \in C$ **and** $(x_l, x_k) \notin Q$ **do**
21: Q $\leftarrow$ Q $\cup \{(x_l, x_k)\}$;
22: **return** TRUE;

In case it finds only one then it will check if $a_i$ is RPC. If it is not then it will be deleted. Function *findTwoSupports* takes as arguments the variables $x_i$ and $x_j$, the value $a_i$, and a parameter $R$, which is set to the single valid residue of $a_i$ in $D(x_j)$ (line 9) or to NIL if none of the two residues is valid.

Function *findTwoSupports* iterates over the values in $D(x_j)$ (line 3). For each value $a_j \in D(x_j)$ it checks if the pair $(a_i, a_j)$ satisfies constraint $c_{ij}$ (this is what function *isConsistent* does). If both residues of $a_i$ in $D(x_j)$ are not valid then after a support is found, the algorithm continues to search for another one. Otherwise, as soon as a support is found that is different than $R$, the function returns having located two supports (lines 9-11).

If only one support $a_j$ is located for $a_i$ then the algorithm checks if the pair $(a_i, a_j)$ is path consistent. During this process it exploits the residues to save redundant work, if possible. Specifically, for any third variable $x_k$ that is constrained with both $x_i$ and $x_j$, we first check if one of the two residues of $a_i$ is valid and if $a_j$ is consistent with that residue (line 16). If this is the case then we know that there is a PC-witness for the pair $(a_i, a_j)$ in $D(x_k)$ without having to iterate over $D(x_k)$. If it is not the case then the check is repeated for the residues of $a_j$ in $D(x_k)$. If we fail to verify the existense of a PC-witness in this way then we iterate over $D(x_k)$ checking if any value $a_k$ is consistent with both $a_i$ and $a_j$. If a PC-witness is found, we proceed with the next variable that is constrained with both $x_i$ and $x_j$. Otherwise, the function returns false, signaling that $a_i$ is not RPC.

**Function 2.** *findTwoSupports*$(x_i, a_i, x_j, R)$:**Boolean**

**if** R = NIL **then** oneSupport $\leftarrow$ FALSE;
**else** oneSupport $\leftarrow$ TRUE;
**for each** $a_j \in D(x_j)$ **do**
  **if** isConsistent$(a_i, a_j)$ **then**
    **if** oneSupport = FALSE **then**
      oneSupport $\leftarrow$ TRUE;
      $\mathrm{R}^1_{x_i,a_i,x_j} \leftarrow a_j$;
    **else**
      **if** $a_j \neq R$ **then**
        $\mathrm{R}^2_{x_i,a_i,x_j} \leftarrow a_j$;
        **return** TRUE;
**if** oneSupport = FALSE **then**
  **return** FALSE
**else**
  **for each** $x_k \in X$, $x_k \neq x_i$ **and** $x_k \neq x_j$, s.t. $(x_k, x_i) \in C$ **and** $(x_k, x_j) \in C$ **do**
    **if** there is a valid residue $\mathrm{R}^*_{x_i,a_i,x_k}$ **and** isConsistent$(R^*_{x_i,a_i,x_k}, a_j)$ **or if** there is a valid residue $\mathrm{R}^*_{x_j,a_j,x_k}$ **and** isConsistent$(R^*_{x_j,a_j,x_k}, a_i)$
    **then continue**;
    PCwitness $\leftarrow$ FALSE;
    **for each** $a_k \in D(x_k)$ **do**
      **if** isConsistent$(a_i, a_k)$ **and** isConsistent$(a_j, a_k)$ **then**
        PCwitness $\leftarrow$ TRUE;
        **break**;
    **if** PCwitness = FALSE **then**
      **return** FALSE;
**return** TRUE;

Moving back to Algorithm 1, if at least one value is deleted from a domain $D(x_i)$, some pairs of variables must be enqueued so that the deletions are propagated. Lines 18-19 enqueue all pairs of the form $(x_k, x_i)$. This ensures that if a value in a domain $D(x_k)$ has lost its last support in $D(x_i)$, it will be processed by the algorithm when the pair $(x_k, x_i)$ is dequeued, and it will be removed. In addition, it ensures that if a value in $D(x_k)$ has been left with only one support in $D(x_i)$, that is not a PC-support, it will be processed and deleted once $(x_k, x_i)$ is dequeued. This means that if we only enqueue pairs of the form $(x_k, x_i)$, we can achieve stronger pruning than AC. However, this is not enough to achieve RPC. We call the version of RPC3 that only enqueues such pairs *restricted RPC3* (rRPC3).

To achieve RPC, for each pair $(x_k, x_i)$ that is enqueued, we also enqueue all pairs of the form $(x_l, x_k)$ s.t. $x_l$ is constrained with $x_i$. This is because after the deletions from $D(x_i)$ the last PC-witness in $D(x_i)$ for some pair of values for variables $x_k$ and $x_l$ may have been deleted. This may cause further deletions from $D(x_l)$.

The worst-case time complexity of RPC3, and rRPC3, is $O(ned^3)$[1]. The space complexity is determined by the space required to store the residues, which is $O(ed)$. The time complexities of algorithms RPC1 and RPC2 are $O(ned^3)$ and $O(ned^2)$ respectively, while their space complexities, for stand-alone use, are $O(ed^2)$ and $O(end)$. RPC3 has a higher time complexity than RPC2, and a lower space complexity than both RPC1 and RPC2. But most importantly, RPC3 does not require the typically quite expensive restoration of data structures after failures when used inside search. In addition, this means that its space complexity remains $O(ed)$ when used inside search, while the space complexities of RPC1 and RPC2 will be even higher than $O(ed^2)$ and $O(end)$.

## 4 Experiments

We have experimented with 17 classes of binary CSPs taken from C.Lecoutre's XCSP repository: *rlfap*, *graph coloring*, *qcp*, *qwh*, *bqwh*, *driver*, *job shop*, *haystacks*, *hanoi*, *pigeons*, *black hole*, *ehi*, *queens*, *geometric*, *composed*, *forced random*, *model B random*. A total of 1142 instances were tested. Details about these classes of problems can be found in C.Lecoutre's homepage. All algorithms used the dom/wdeg heuristic for variable ordering [6] and lexicographic value ordering. The experiments were performed on a FUJITSU Server PRIMERGY RX200 S7 R2 with Intel(R) Xeon(R) CPU E5-2667 clocked at 2.90GHz, with 48 GB of ECC RAM and 16MB cache.

We have compared search algorithms that apply rRPC3 (resp. RPC3) during search to a baseline algorithm that applies AC (i.e. MAC) and also to an algorithm that applies lmaxRPC. AC and lmaxRPC were enforced using the $AC^r$ and lmaxRPC3 algorithms respectively. For simplicity, the four search algorithms will be denoted by AC, rRPC, RPC, and maxRPC hereafter. Note that a MAC algorithm with $AC^r$ and dom/wdeg for variable ordering is considered as the best general purpose solver for binary CSPs.

A timetout of 3600 seconds was imposed on all four algorithms for all the tested instances. Importantly, **rRPC only timed out on instances where AC and maxRPC also timed out**. On the other hand, there were several cases where rRPC finished search within the time limit but one (or both) of AC and maxRPC timed out. There were a few instances where RPC timed out while rRPC did not, but the opposite never occured.

Table 1 summarizes the results of the experimental evaluation for specific classes of problems. For each class we give the following information:

- The mean node visits and run times from non-trivial instances that were solved by all algorithms within the time limit. We consider as trivial any instance that was solved by all algorithms in less than a second.
- Node visits and run time from the single instance where AC displayed its best performance compared to rRPC.

[1] The proof is quite simple but it is omitted for space reasons.

- Node visits and run time from the single instance where maxRPC displayed its best performance compared to rRPC.
- Node visits and run time from a representative instance where rRPC displayed good performance compared to AC, excluding instances where AC timed out.
- The number of instances where AC, RPC, maxRPC timed out while rRPC did not. This information is given only for classes where at least one such instance occured.
- The number of instances where AC, rRPC, RPC, or maxRPC was the winning algorithm, excluding trivial instances.

Comparing AC to rRPC we can note the following. rRPC is more efficient in terms of mean run time performance on all classes of structured problems with the exception of *queens*. The difference in favor of rRPC can be quite stunning, as in the case of *qwh* and *qcp*. The numbers of node visits in these classes suggest that rRPC is able to achieve considerable extra pruning, and this is then reflected on cpu times.

Importantly, in all instances of 16 classes (i.e. all classes apart from *queens*) AC was at most 1.7 times faster than rRPC. In contrast, there were 7 instances from *rlfap* and 12 from *graph coloring* where AC timed out while rRPC finished within the time limit. The mean cpu time of rRPC on these *rlfap* instances was 110 seconds while on the 12 *graph coloring* instances the cpu time of rRPC ranged from 1.8 to 1798 seconds. In addition, there were numerous instances where rRPC was orders of magnitude faster than AC. This is quite common in *qcp* and *qwh*, as the mean cpu times demonstrate, but such instances also occur in *graph coloring*, *bqwh*, *haystacks* and *ehi*.

Regarding random problems, rRPC achieves similar performance to AC on *geometric* (which is a class with some structure) and is slower on *forced random* and *model B random*. However, the differences in these two classes are not significant. The only class where there are significant differences in favour of AC is *queens*. Specifically, AC can be up to 5 times faster than rRPC on some instances, and orders of magnitude faster than both RPC and maxRPC. This is because all algorithms spend a lot of time on propagation but evidently the strong local consistencies achieve little extra pruning. Importantly, the low cost of rRPC makes its performance reasonable compared to the huge run times of RPC and maxRPC.

The comparison between RPC and AC follows the same trend as that of rRPC and AC, but importantly the differences in favour of RPC are not as large on structured problems where AC is inefficient, while AC is clearly faster on random problems, and by far superior on dense structured problems like *queens* and *pigeons*.

Comparing the mean performance of rRPC to RPC and maxRPC we can note that rRPC is more efficient on all classes. There are some instances where RPC or/and maxRPC outperform rRPC due to their stronger pruning, but the differences in favour of RPC and maxRPC are rarely significant. In contrast, rRPC can often be orders of magnitude faster. An interesting observation that

**Table 1.** Node visits (n), run times in secs (t), number of timeouts (#TO) (if applicable), and number of wins (winner) in summary. The number in brackets after the name of each class gives the number of instances tested.

| class | AC | | rRPC | | RPC | | maxRPC | |
|---|---|---|---|---|---|---|---|---|
| | (n) | (t) | (n) | (t) | (n) | (t) | (n) | (t) |
| **rlfap** (24) | | | | | | | | |
| *mean* | 29045 | 64.1 | 11234 | 32.3 | 10878 | 39.0 | 8757 | 134.0 |
| *best AC* | 12688 | 9.3 | 11813 | 14.5 | 10048 | 18.2 | 5548 | 33.2 |
| *best maxRPC* | 8405 | 10.1 | 3846 | 4.4 | 3218 | 6.86 | 1668 | 4.8 |
| *good rRPC* | 19590 | 28.8 | 5903 | 8.2 | 5197 | 10.4 | 8808 | 23.7 |
| *#TO* | 7 | | | | 3 | | 6 | |
| *winner* | 1 | | 18 | | 1 | | 0 | |
| **qcp** (60) | | | | | | | | |
| *mean* | 307416 | 345.4 | 37725 | 44.8 | 43068 | 167.1 | 49005 | 101.3 |
| *best AC* | 36498 | 63.5 | 36286 | 73.7 | 57405 | 354.3 | 63634 | 173.5 |
| *best maxRPC* | 20988 | 16.8 | 7743 | 7.6 | 4787 | 11.9 | 1723 | 1.8 |
| *good rRPC* | 1058477 | 761 | 65475 | 53.5 | 67935 | 162.9 | 54622 | 63.1 |
| *winner* | 2 | | 8 | | 0 | | 4 | |
| **qwh** (40) | | | | | | | | |
| *mean* | 205232 | 1348.2 | 20663 | 46.2 | 28694 | 177.9 | 24205 | 64.7 |
| *best AC* | 6987 | 4.5 | 3734 | 3.0 | 5387 | 11.9 | 3174 | 3.2 |
| *best maxRPC* | 231087 | 461.4 | 30926 | 72.3 | 30434 | 187.6 | 13497 | 35.8 |
| *good rRPC* | 445771 | 859.6 | 35923 | 79.5 | 56965 | 375.4 | 37582 | 103.9 |
| *winner* | 0 | | 9 | | 0 | | 6 | |
| **bqwh** (200) | | | | | | | | |
| *mean* | 28573 | 7.6 | 7041 | 2.4 | 5466 | 2.9 | 6136 | 2.7 |
| *best AC* | 5085 | 1.2 | 4573 | 1.4 | 4232 | 2.0 | 3375 | 1.3 |
| *best maxRPC* | 324349 | 85.3 | 122845 | 46.1 | 56020 | 33.8 | 64596 | 29.3 |
| *good rRPC* | 83996 | 22.5 | 7922 | 2.6 | 10858 | 5.6 | 9325 | 4.2 |
| *winner* | 2 | | 36 | | 13 | | 20 | |
| **graph coloring** (177) | | | | | | | | |
| *mean* | 322220 | 88.6 | 261882 | 60.4 | 192538 | 73.7 | 263227 | 138.79 |
| *best AC* | 1589650 | 442.5 | 1589650 | 743.8 | 1266416 | 930.8 | 1589650 | 1010.0 |
| *best maxRPC* | 1977536 | 647.9 | 1977536 | 762.4 | 1265930 | 613.7 | 1977536 | 759.8 |
| *good rRPC* | 31507 | 189.1 | 3911 | 15.6 | 2851 | 18.2 | 10477 | 62.7 |
| *#TO* | 12 | | | | 1 | | 5 | |
| *winner* | 8 | | 35 | | 17 | | 0 | |
| **geometric** (100) | | | | | | | | |
| *mean* | 111611 | 58.4 | 54721 | 58.6 | 52416 | 97.3 | 38227 | 190.9 |
| *best AC* | 331764 | 203.1 | 169871 | 218.1 | 160428 | 358.1 | 113878 | 696.2 |
| *best maxRPC* | 67526 | 28.1 | 31230 | 28.1 | 30229 | 53.5 | 20071 | 73.6 |
| *good rRPC* | 254304 | 123.3 | 119248 | 117.4 | 117665 | 203.0 | 84410 | 363.3 |
| *winner* | 12 | | 11 | | 1 | | 0 | |
| **forced random** (20) | | | | | | | | |
| *mean* | 348473 | 143.5 | 197994 | 177.2 | 191114 | 309.8 | 154903 | 455.4 |
| *best AC* | 1207920 | 491.5 | 677896 | 596.7 | 654862 | 1040.4 | 538317 | 1551.4 |
| *best maxRPC* | 26729 | 7.6 | 12986 | 9.0 | 12722 | 13.5 | 9372 | 22.4 |
| *good rRPC* | 455489 | 201.6 | 270345 | 258.5 | 262733 | 462.0 | 207267 | 651.2 |
| *winner* | 20 | | 0 | | 0 | | 0 | |
| **model B random** (40) | | | | | | | | |
| *mean* | 741124 | 194.3 | 383927 | 224.5 | 361926 | 346.5 | 28871 | 1044.9 |
| *best AC* | 2212444 | 669.8 | 1197369 | 805.7 | 1136257 | 1283.8 | - | TO |
| *best maxRPC* | 345980 | 81.2 | 181127 | 94.4 | 171527 | 142.7 | 130440 | 405.8 |
| *good rRPC* | 127567 | 32.4 | 43769 | 24.4 | 41328 | 39.1 | 51455 | 171.5 |
| *#TO* | 0 | | | | 0 | | 14 | |
| *winner* | 39 | | 1 | | 0 | | 0 | |
| **queens** (14) | | | | | | | | |
| *mean* | 2092 | 14.2 | 797 | 59.2 | 2476 | 1032.2 | 953 | 2025.2 |
| *best AC* | 150 | 9.2 | 149 | 43.0 | 149 | 499.1 | 149 | 3189.2 |
| *best maxRPC* | 7886 | 10.8 | 2719 | 11.5 | 9425 | 910.5 | 3341 | 367.9 |
| *good rRPC* | 7886 | 10.8 | 2719 | 11.5 | 9425 | 910.5 | 3341 | 367.9 |
| *#TO* | 0 | | | | 0 | | 1 | |
| *winner* | 4 | | 0 | | 0 | | 0 | |

requires further investigation is that in some cases the node visits of rPRC are fewer than RPC and and/or maxRPC despite the weaker pruning. This usually occurs on soluble instances and suggests that the interaction with the dom/wdeg heuristic can guide search to solutions faster.

Finally, the classes not shown in Table 1 mostly include instances that are either very easy or very hard (i.e. all algorithms time out). Specifically, instances in *composed* and *hanoi* are all trivial, and the ones in *black hole* and *job shop* are either trivial or very hard. Instances in *ehi* typically take a few seconds for AC and under a second for the other three algorithms. Instances in *haystacks* are very hard except for a few where AC is clearly outperformed by the other three algorithms. For example, in *haystacks-04* AC takes 8 seconds and the other three take 0.2 seconds. Instances in *pigeons* are either trivial or very hard except for a few instances where rRPC is the best algorithm followed by AC. For example on *pigeons-12* AC and rRPC take 709 and 550 seconds respectively, while RPC and maxRPC time out. Finally, *driver* includes only 7 instances. Among them, 3 are trivial, rRPC is the best algorithm on 3, and AC on 1.

## 5 Conclusion

RPC was recognized as a promising alternative to AC but has been neglected for the past 15 years or so. In this paper we have revisited RPC by proposing RPC3, a new algorithm that utilizes ideas, such as residues, that have become standard in recent years when implementing AC or maxRPC algorithms. Using RPC3 and a restricted variant we performed the first wide experimental study of RPC when used inside search. Perhaps surprisingly, results clearly demostrate that rRPC3 is by far more efficient than state-of-the-art AC and maxRPC algorithms when applied during search. This challenges the common perception that MAC is the best general purpose solver for binary CSPs.

## References


1. Balafoutis, T., Paparrizou, A., Stergiou, K., Walsh, T.: New algorithms for max restricted path consistency. Constraints **16**(4), 372–406 (2011)
2. Balafrej, A., Bessiere, C., Bouyakh, E., Trombettoni, G.: Adaptive singleton-based consistencies. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, pp. 2601–2607 (2014)
3. Barták, R., Erben, R.: A new algorithm for singleton arc consistency. In: Proceedings of the Seventeenth International Florida Artificial Intelligence, pp. 257–262 (2004)
4. Berlandier, P.: Improving domain filtering using restricted path consistency. In: Proceedings of IEEE CAIA 1995, pp. 32–37 (1995)
5. Bessiere, C., Cardon, S., Debruyne, R., Lecoutre, C.: Efficient Algorithms for Singleton Arc Consistency. Constraints **16**, 25–53 (2011)
6. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI 2004, Valencia, Spain (2004)

7. Debruyne, R., Bessière, C.: From restricted path consistency to max-restricted path consistency. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 312–326. Springer, Heidelberg (1997)
8. Debruyne, R., Bessière, C.: Domain Filtering Consistencies. JAIR **14**, 205–230 (2001)
9. Grandoni, F., Italiano, G.F.: Improved algorithms for max-restricted path consistency. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 858–862. Springer, Heidelberg (2003)
10. Lecoutre, C., Hemery, F.: A study of residual supports in arc cosistency. In: Proceedings of IJCAI 2007, pp. 125–130 (2007)
11. Lecoutre, C., Prosser, P.: Maintaining singleton arc consistency. In: 3rd International Workshop on Constraint Propagation and Implementation (CPAI 2006), pp. 47–61 (2006)
12. Likitvivatanavong, C., Zhang, Y., Bowen, J., Shannon, S., Freuder, E.: Arc consistency during search. In: Proceedings of IJCAI 2007, pp. 137–142 (2007)
13. Prosser, P., Stergiou, K., Walsh, T.: Singleton consistencies. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 353–368. Springer, Heidelberg (2000)
14. Vion, J., Debruyne, R.: Light algorithms for maintaining max-RPC during search. In: Proceedings of SARA 2009 (2009)