On Neighborhood Singleton Consistencies

Anastasia Paparrizou
CRIL-CNRS and Université d'Artois
Lens, France
paparrizou@cril.fr

Kostas Stergiou
University of Western Macedonia
Kozani, Greece
kstergiou@uowm.gr

Abstract

CP solvers predominantly use arc consistency (AC) as the default propagation method. Many stronger consistencies, such as triangle consistencies (e.g. RPC and maxRPC) exist, but their use is limited despite results showing that they outperform AC on many problems. This is due to the intricacies involved in incorporating them into solvers. On the other hand, singleton consistencies such as SAC can be easily crafted into solvers but they are too expensive. We seek a balance between the efficiency of triangle consistencies and the ease of implementation of singleton ones. Using the recently proposed variant of SAC called Neighborhood SAC as basis, we propose a family of weaker singleton consistencies. We study them theoretically, comparing their pruning power to existing consistencies. We make a detailed experimental study using a very simple algorithm for their implementation. Results demonstrate that they outperform the existing propagation techniques, often by orders of magnitude, on a wide range of problems.

1 Introduction

Recent works on local consistencies stronger than arc consistency (AC) have shown that they can quite often outperform it when used inside search. Two classes of strong local consistencies for binary constraints have received the most attention. The first class, inspired by path consistency, includes triangle-based consistencies like restricted path consistency (RPC) [Berlandier, 1995], path inverse consistency (PIC) [Freuder and Elfe, 1996], and max restricted path consistency (maxRPC) [Debruyne and Bessiere, 1997]. The other class is that of singleton consistencies, with singleton arc consistency (SAC) being the prime example [Debruyne and Bessiere, 2001]. Recently, a variant of SAC called neighborhood SAC (NSAC) was proposed [Wallace, 2015; 2016a]. The difference between NSAC and SAC is that at each singleton check, i.e. temporary assignment of a variable x, NSAC restricts the application of AC to the neighborhood of x, whereas SAC applies AC to the whole problem.

Experiments demostrate that triangle-based methods, specifically RPC and maxRPC, are quite competitive to AC when maintained, and very often outperform it [Vion and Debruyne, 2009; Balafoutis *et al.*, 2011; Stergiou, 2015]. On the other hand, the full application of SAC throughout search is way too expensive despite recent developments in SAC algorithms. Hence only its selective application is a viable option, and only for specific problems [Bessiere *et al.*, 2011; Balafrej *et al.*, 2014]. Overall, it seems that triangle-based consistencies are a much better option than singleton ones as alternatives for AC. Some evidence about this can also be found in [Wallace, 2016b]. However, a significant advantage that singleton consistencies have, is that they can be quite easily integrated in CP solvers.

In this paper we seek a balance between the efficiency of triangle-based consistencies and the ease of implementation of singleton ones. To achieve this we study a number of new singleton consistencies that are based on NSAC and follow the reasoning behind either RPC or maxRPC. These consistencies are much cheaper than SAC while at the same time being very easy to implement.

We begin by considering singleton consistencies inspired by maxRPC. We first show that NSAC achieves a level of local consistency that is strictly stronger than that achieved by maxRPC. Then we consider a weaker version of NSAC, called NS1pAC, that only applies one pass of AC in the neighborhood of the considered variable during a singleton check. We show that this property is incomparable to maxRPC. However, if we simply insist that the constraints involving the considered variable are examined first during the single AC pass, then NS1pAC is strictly stronger than maxRPC, and interestingly it can be applied with the same asymptotic cost.

Then we turn our attention to singleton consistencies inspired by RPC. During a singleton check these consistencies are enforced by first applying AC on the constraints involving the considered variable x. Then, following the reasoning of RPC, the domain sizes of the variables in the neighborhood of x are inspected. If at least one of these variables has a singleton domain then AC is applied, either in the entire neighborhood of x or in a sub-graph of the neighborhood, depending of the particular method. If there is no singleton domain in the neighborhood of x then nothing is done. Results from a theoretical analysis of these RPC-inspired methods show that their pruning capabilities lay between RPC and NSAC.

Finally, we make an experimental evaluation of all the considered methods. The results outline our most important con-

tribution: Singleton neighborhood consistencies are not only very easy to implement, but also significantly outperform all of the competitive existing methods (AC, RPC, maxRPC) on numerous problems, and quite often by many orders of magnitude, while being less frequently outperformed by large margins. This is the first time where the practical value of fully maintaining singleton consistencies during search is demostrated on a wide range of problems.

2 Background

A binary *Constraint Satisfaction Problem* (CSP) $\mathcal{P}$ 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 finite domains, one for each variable, with maximum cardinality d, $\mathcal{C} = \{c_1, \ldots, c_e\}$ is a set of e constraints. A binary constraint c_{ij} involves variables x_i and x_j and specifies the allowed combinations of values for the two variables.

A binary CSP $\mathcal{P}$ is typically depicted as graph $\mathcal{G}$ where variables correspond to nodes and constraints to edges. A variable x_j is a *neighbor* of a variable x_i iff $c_{ij} \in \mathcal{C}$. The neighborhood N(x_i) of a variable x_i is the subgraph of $\mathcal{G}$ that includes x_i, all neighbors of x_i, any constraint between x_i and one of its neighbors, and any constraint between two neighbors of x_i.

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 a *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 $\mathcal{D}$ and all the variables in $\mathcal{X}$ are AC.

A number of weaker variants of AC have been proposed and were quite often used in the past. One such variant is 1-pass AC (aka *full look-ahead* when used inside search). This method considers each pair of variables only once and therefore removes arc inconsistent values that can be detected by making only one pass through the constraints of the problem.

Stronger methods based on AC, such as singleton AC, have also been considered. A value $a_i \in D(x_i)$ is *singleton AC* (SAC) iff after restricting $D(x_i)$ to a_i and applying AC to $\mathcal{P}$, there is no domain wipeout (DWO) [Debruyne and Bessiere, 2001]. A problem is SAC iff all values in all domains are SAC. *Neighborhood SAC* (NSAC) is a variant of SAC which, after restricting $D(x_i)$ to a_i, applies AC to N(x_i).

Another widely known local consistency is path consistency. 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 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 $D(x_k)$.

If AC, or (N)SAC, is enforced on a problem then single inconsistent values can be identified and removed from the corresponding domains. These are examples of *domain filtering* local consistencies. In contrast, if PC is enforced then pairs of inconsistent values can be identified. To store the derived knowledge, new binary constraints must be introduced or existing ones must be modified, but this can be tricky and non-intuitive. Hence, maxRPC, RPC and other domain filtering variants of PC have been proposed.

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 PC. 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 PC.

It has been shown that a weaker variant of maxRPC (resp. RPC), called lmaxRPC (resp. lRPC), is more cost effective in practice [Vion and Debruyne, 2009; Balafoutis *et al.*, 2011; Stergiou, 2015]. This method enforces a restriction on how the deletion of a value that is not maxRPC (resp. RPC) is propagated by only checking for PC-support loss after a deletion and avoiding to check for PC-witness loss.

Following [Debruyne and Bessiere, 2001], a consistency property A is *stronger* than B iff in any problem in which A holds then B holds, and *strictly stronger* iff there is at least one problem in which B holds but A does not. A local consistency property A is incomparable with B iff A is not stronger than B nor vice versa.

3 NSAC and maxRPC

In this section we study the relationship between NSAC and maxRPC. Let us first define a weaker variant of NSAC, which is based on 1-pass AC.

Definition 1 A value $a_i \in D(x_i)$ is *neighborhood singleton 1-pass arc consistent* (NS1pAC) iff, given an order of the variables in N(x_i), after restricting $D(x_i)$ to a_i and applying 1-pass AC to N(x_i) according to the order, there is no domain wipeout (DWO). A problem is NS1pAC iff all values in all domains are NS1pAC.

Given that the only difference between NSAC and NS1pAC is that the former applies full AC in the neighborhood of the considered variable during a singleton check while the latter applies AC in one pass, it is obvious that NSAC is strictly stronger than NS1pAC. Importantly, the amount of pruning achieved by 1-pass AC depends on the order in which variables are considered. Inadvertently, this affects the pruning strength of NS1pAC.

We show that in the general case, because of this effect that the ordering has, NS1pAC is incomparable to maxRPC. We assume that the algorithm for 1-pass AC visits the variables one by one, and for each visited variable x it removes from the domain of each other variable any value that has no support in $D(x)$.

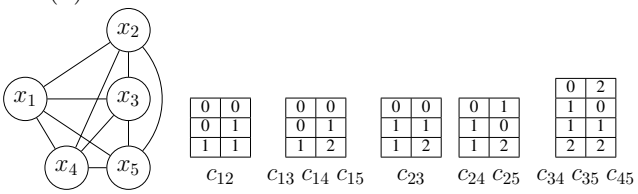

Figure 1: A problem that is maxRPC but not NS1pAC.

Proposition 1 NS1pAC is incomparable to maxRPC.

Proof: For a problem that is maxRPC but not NS1pAC, consider Figure 1. The domains of the variables are: $D(x_1) = D(x_2) = \{0, 1\}$, $D(x_3) = D(x_4) = D(x_5) =$

$\{0, 1, 2\}$. The tables show the allowed pairs of values for the constraints. Value 0 of x_1 is maxRPC because we can find a PC-support in each of the other variables. Now assume that NS1pAC assigns 0 to x_1 and then applies 1-pass AC considering the variables in the order x_1, x_5, x_4, x_3, x_2. This will result in a DWO for x_2. Hence, value 0 of x_1 is not NS1pAC.

For a problem that is NS1pAC but not maxRPC, consider Figure 2. Assume that $D(x_1) = \{0, 1\}$, $D(x_2) = D(x_3) = D(x_4) = \{0, 1, 2\}$. Value 0 of x_1 is not maxRPC because it has no PC-support in any of the other variables. However, if we temporarily assign 0 to x_1 and apply 1-pass AC in the order x_2, x_3, x_4, x_1, then no pruning will occur until x_1 is visited, in which case value 2 will be deleted from the domains of x_2, x_3, x_4. But these deletions will not be propagated, since no constraint will be re-examined. Hence there will be no DWO meaning that 0 is NS1pAC. ■

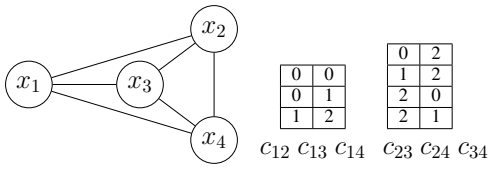

Figure 2: A problem that is NS1pAC but not maxRPC.

It is easy to see that if lmaxRPC is used in the second part of the proof instead of maxRPC, still value 0 of x_1 will be deleted. This means that NS1pAC is not only incomparable to maxRPC, but also to lmaxRPC.

But under a simple condition which *partially* specifies the order in which variables are examined when 1-pass AC is enforced during a singleton check, NS1pAC is strictly stronger than maxRPC.

Condition *FC* Every time a singleton check is performed on a value $a_i \in D(x_i)$, all constraints involving x_i are first processed, and any value in a domain $D(x_j)$ that is not supported by a_i is pruned. Then all other variables in N(x_i) are processed.

Given that at a singleton check value a_i is (temporarily) assigned to x_i, the above condition simply specifies that the algorithm used for 1-pass AC first performs a type of *forward checking* between a_i and all variables constrained with x_i. Then 1-pass AC is applied on all other constraints.

Proposition 2 Under Condition *FC*, NS1pAC is strictly stronger than maxRPC.

Proof: Assume that a value $a_i \in D(x_i)$ is not maxRPC. This means that it has no PC-support on some variable x_j. Since Condition *FC* is applied, after a_i is assigned to x_i, any value that is not a support for a_i will be deleted from the domains of all of x_i's neighbors, including x_j. Now take some value $b_j \in D(x_j)$ that supports a_i. Since b_j is not a PC-support for a_i, there exists a variable x_k in whose domain no value is a support for both a_i and b_j. Since, after Condition *FC*, $D(x_k)$ only contains values that support a_i and none of them supports b_j, when 1-pass AC examines x_k it will delete b_j. Using the same reasoning, it will also delete any other value in $D(x_j)$. Hence, we will have a DWO, meaning that a_i is not NS1pAC. For strictness consider again the first part of the proof of Proposition 1. ■

In the following when referring to NS1pAC we will mean the version of NS1pAC which applies Condition *FC*. It is now trivial to prove that NSAC is strictly stronger than maxRPC.

Corollary 1 NSAC is strictly stronger than maxRPC.

Proof: To check if a value $a_i \in D(x_i)$ is NSAC, AC must be applied after the assignment of a_i to x_i is made. This encompasses the appplication of Condition *FC* and achieves at least the same pruning as 1-pass AC . Hence, trivially from Proposition 2 NSAC is strictly stronger than maxRPC. ■

4 Singleton Consistencies Inspired by RPC

We now turn our attention to RPC by considering NSAC-based local consistencies that follow the reasoning behind RPC. We define two local consistencies as well as their 1-pass variants, and study their pruning power.

4.1 Restricted Neighborhood and sub-Neighborhood SAC

Before formally defining the new local consistencies, let us briefly describe how they are applied, assuming that a variable x is singleton checked using them. We call these consistencies *Restricted NSAC* (RNSAC) and *Restricted sub-Neighborhood SAC* (RsNSAC).

- At each singleton check $x_i = a_i$, first Condition *FC* is applied.
- If after the application of Condition *FC* at least one variable in N(x_i) has a singleton domain then, in the case of RNSAC, AC is applied in N(x_i), while in the case of RsNSAC, AC is applied in a sub-graph of N(x_i). If a DWO is detected then a_i is removed.
- If after the application of Condition *FC* there is no singleton domain in N(x_i) then nothing is done (AC is not further applied).

The sub-graph of N(x_i) where AC is applied in the case of RsNSAC is specified as follows: Assuming that $SD(x_i)$ is the set of variables with singleton domains after Condition *FC* has been applied, this sub-graph of N(x_i) includes x_i and any variable that belongs to $SD(x_i)$ or is constrained with a variable in $SD(x_i)$.

Both RNSAC and RsNSAC are inspired by RPC which tries to extend a pair of values $a_i \in D(x_i)$ and $b_j \in D(x_j)$ to variables constrained with x_i and x_j only if b_j is the single support of a_i in $D(x_j)$.

More formally, RNSAC is defined as follows.

Definition 2 A value $a_i \in D(x_i)$ is *restricted neighborhood singleton arc consistent* (RNSAC) iff it is AC and the following holds. If after restricting $D(x_i)$ to a_i and applying Condition *FC* there is at least one variable in N(x_i) with singleton domain then the application of AC to N(x_i) does not result in a DWO.

To define RsNSAC formally, we need the following.

Definition 3 Let $SD(x_i)$ be the set of variables in N(x_i) that have singleton domains after the application of Condition *FC* during the singleton check of a variable x_i. Then $G_{SD(x_i)}$ is the sub-graph of N(x_i) that includes x_i, the variables in $SD(x_i)$, any variable in N(x_i) constrained with a variable in $SD(x_i)$, and any constraint that involves a variable in $SD(x_i)$ and a variable in N(x_i).

The difference between N(x_i) and $G_{SD(x_i)}$ is that the former includes constraints that involve two variables that do not belong to $SD(x_i)$ while the latter excludes such constraints. For example, consider the problem in Figure 3a. If we apply RsNSAC after 0 is assigned to x_1, then after Condition *FC*, $D(x_2)$ will be the only variable with a singleton domain (i.e. $SD(x_1)$=$\{x_2\}$). So AC will be applied in the subgraph $G_{SD(x_1)}$ comprising x_1, x_2, and x_3, which is constrained with x_2, but not x_4, as it is not constrained with x_2. Hence, c_{34} will not be considered.

Definition 4 A value $a_i \in D(x_i)$ is *restricted subneighborhood singleton arc consistent* (RsNSAC) iff it is AC and the following holds. If after restricting $D(x_i)$ to a_i and applying Condition *FC* there is at least one variable in N(x_i) with singleton domain then the application of AC to $G_{SD(x_i)}$ does not result in a DWO.

We call the corresponding properties that apply 1-pass AC RNS1pAC and RsNS1pAC.

4.2 Theoretical Results

By definition RNSAC (resp. RNS1pAC) is strictly stronger than RsNSAC (resp. RsNS1pAC). Accordingly, RNSAC and RNS1pAC are strictly weaker than NSAC and NS1pAC respectively. We now show that RsNSAC and RsNS1pAC are strictly stronger than RPC.

Proposition 3 RsNSAC and RsNS1pAC are strictly stronger than RPC.

Proof: We prove that RsNS1pAC is strictly stronger than RPC. By definition, it also holds for RsNSAC. Assume that a value $a_i \in D(x_i)$ is not RPC. This means that either it is not AC or it has a single support b_j on some variable x_j and the pair (a_i,b_j) is not PC. In the former case RsNS1pAC will delete a_i because when Condition *FC* applied, a DWO will occur. In the latter case, after RsNS1pAC assigns a_i to x_i, the application of Condition *FC* will delete from the domains of all of x_i's neighbors, including x_j, any value that is not a support for a_i. This will leave b_j as the only value in $D(x_j)$. Assume that x_j is the only variable in N(x_i) that is left with a singleton domain after Condition *FC* has been applied. As a result, 1-pass AC will be applied in the sub-network involving x_i, x_j, and any variable in N(x_i) that is constrained with x_j. As the pair (a_i,b_j) is not path consistent, there must be a variable x_k in N(x_i) that is constrained with x_j, in whose domain no value is a support for both a_i and b_j. As $D(x_k)$ only contains values that support a_i, when x_k is considered by 1-pass AC, b_j will be deleted and $D(x_j)$ will be wipped out, meaning that a_i is not RsNS1pAC.

For strictness consider the problem in Figure 3a. Assume that we are examining whether value 0 of x_1 is RPC. The domains of the variables are as follows: $D(x_1) = \{0, 1\}$, $D(x_2) = D(x_3) = D(x_4) = \{0, 1, 2\}$. The tables show the allowed pairs of values for the constraints in the problem. Value 0 of x_1 is RPC because it is AC, it has more than one support in $D(x_3)$ and $D(x_4)$, and its single support 0 in $D(x_2)$ is also a PC-support. On the other hand, the application of RsNS1pAC will first assign 0 to x_1 and enforce Condition *FC*. This will remove value 2 from $D(x_3)$ and $D(x_4)$ and will leave x_2 with only value 0 in its domain, meaning that 1-pass AC will be next applied. Assuming that the variables are considered in the order x_2, x_3, x_4, no value removal will occur when x_2 is considered. When x_3 is considered, both 0 and 1 will be removed from $D(x_4)$, which will therefore be wipped out. Hence, value 0 of x_1 is not RsNS1pAC and will be deleted. ■

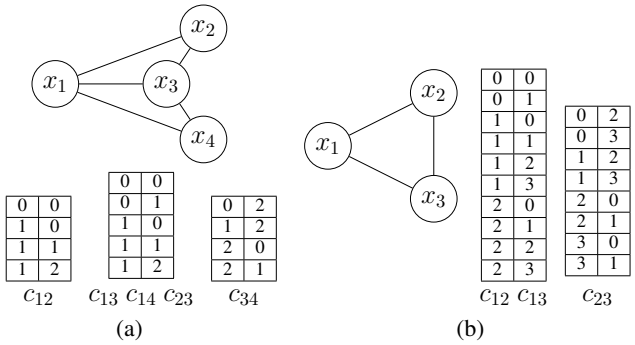

Figure 3: (a) A problem that is RPC but not RsNS1pAC. (b) A problem that is RNSAC but not maxRPC.

We now compare RNSAC, RsNSAC, and their 1-pass versions to maxRPC and lmaxRPC.

Proposition 4 RNSAC, RNS1pAC, RsNSAC, and RsNS1pAC are incomparable to maxRPC and lmaxRPC.

Proof: For a problem that is (l)maxRPC but not RsNS1pAC, which is the weakest among the 4 singleton consistencies, consider Figure 3a. Value 0 of x_1 has a PC-support in each of the other variables, and therefore it is (l)maxRPC. But as explained above, it is not RsNS1pAC.

Now consider the problem of Figure 3b. The domains are: $D(x_1) = \{0, 1, 2\}$, $D(x_2) = D(x_3) = \{0, 1, 2, 3\}$. This problem is RNSAC, which is the strongest among the 4 singleton consistencies, because it is AC and each value of each variable has at least two supports in each of the other two variables. However, value 0 of x_1 is not (l)maxRPC because it has no PC-support in $D(x_2)$. ■

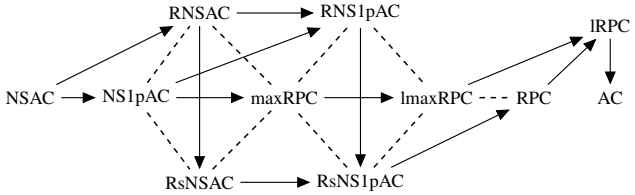

Figure 4: A solid line denotes the "stronger than" relationship. A dashed line denotes the "incomparable" relationship.

Figure 4 summarizes the relationships between the various local consistencies discussed, with respect to their pruning power. We include some relationships that have not been

proved above but are very easy to prove. But to make the figure easier to read, some other relationships are omitted.

5 Algorithm and Complexities

As shown by Wallace, any SAC algorithm can be used as basis to build an NSAC algorithm. However, SAC algorithms such as SAC-SDS, SAC-Opt, and SAC-3, described in [Bessiere *et al.*, 2011], are quite complex, and in the end NSAC algorithms based on them are inferior in cpu times to the simple NSACQ algorithm of [Wallace, 2016a].

Algorithm 1 The RNSQ algorithm

```
1: initialize(Q)
2: while Q ≠ ∅ do
3:     select and remove a variable x_i from Q;
4:     Deletion ← FALSE;
5:     for each a_i ∈ D(x_i) do
6:         assign a_i to x_i and apply Condition FC;
7:         if a DWO is detected then
8:             remove a_i from D(x_i);
9:             Deletion ← TRUE;
10:        else
11:            if a neighbor of x_i has singleton domain then
12:                apply AC to N(x_i);
13:                if a DWO is detected then
14:                    remove a_i from D(x_i);
15:                    Deletion ← TRUE;
16:        if D(x_i) = ∅ then
17:            return FALSE;
18:    if Deletion = TRUE then
19:        add to Q any x_j s.t. x_j ∈ N(x_i);
20: return TRUE;
```

We now give the algorithm we used in our experiments. We present it for the case of RNSAC, which as shown at the next section, is the best among all methods, calling it RNSQ. It follows the reasoning behind NSACQ (i.e. the use of a queue to propagate deletions) and requires minimal changes to apply any of the other consistencies studied. Like NSACQ, RNSQ is a very simple algorithm stripped of any optimizations.

Algorithm 1 uses a data structure Q implemented as a queue. For preprocessing, Q is initialized with all variables in the problem. For use inside search, Q is initialized with all neighbors of the currently instantiated variable. During the singleton check of a value $a_i \in D(x_i)$, Condition *FC* is first applied. Then if there is no DWO and one of x_i's neighbors is left with a singleton domain, AC in applied in $\mathrm{N}(x_i)$. If a DWO is detected, a_i is deleted. It is is easy to see that if line 11 is omitted, the algorithm achieves NSAC.

The worst-case time complexity of applying RNSAC or NSAC using Algorithm 1 is $\mathrm{O}(en^2d^5)$ assuming that AC-3 or one of its variants is used to apply AC in line 12. If an optimal AC algorithm is used then the cost of NSAC falls by a factor of d. But it is nowdays accepted that AC3-like algorithms are more efficient than optimal ones, especially when used inside search, because of their light use of data structures [Lecoutre and Hemery, 2007; Likitvivatanavong *et al.*, 2007]. The complexity of applying NS1pAC and RNS1pAC is $\mathrm{O}(en^2d^4)$ because only one pass is made through the constraints. Interestingly, this is the same as the complexity of maxRPC3^{rm} which is the best maxRPC algorithm [Balafoutis *et al.*, 2011], and it is slightly higher than the $\mathrm{O}(end^4)$ complexity of lmaxRPC3^{rm}. Let us not forget though that NS1pAC is strictly stronger than maxRPC (and thus also lmaxRPC) when Condition *FC* holds.

6 Experiments

We experimented with 16 classes of binary CSPs taken from C.Lecoutre's XCSP repository: *rlfap*, *graph coloring*, *qcp*, *qwh*, *bqwh*, *driver*, *haystacks*, *hanoi*, *pigeons*, *black hole*, *ehi*, *queens*, *queensAttacking*, *queensKnights*, *geometric*, *composed*. A total of 1054 instances were tested. We mainly used dom/ddeg instead of the more efficient dom/wdeg to avoid severe interference between the heuristic and propagation.

The experiments were performed on a FUJITSU Server (2.90GHz, 48 GB RAM, 16MB cache). We compared search algorithms that maintain NSAC and its variants to ones that maintain AC, lRPC, and lmaxRPC. The three baseline methods were implemented using the corresponding state-of-the-art algorithms [Lecoutre and Hemery, 2007; Balafoutis *et al.*, 2011; Stergiou, 2015]. For simplicity, the three search algorithms will be denoted by AC, lRPC, and lmaxRPC hereafter. A timeout of 3600 seconds was imposed on all algorithms.

Table 1 summarizes the results for specific classes of problems, comparing NSAC, NS1pAC, RNSAC, RNS1pAC, and RsNSAC to AC, lRPC, lmaxRPC. For the 1-pass methods AC is applied following the lexicographic order. For each class we give the following data: 1) 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. 2) The number of timeouts for each algorithm, excluding instances where all algorithms timed out.

Table 1 shows that NSAC and its variants are clearly more efficient than the existing propagation methods. Specifically, there are classes where the former methods are by far faster (*qcp*, *qwh*, *bqwh*), others where they are very competitive (*coloring*), and others where they overwhelmingly dominate (*composed-25-10*, *queensKnights*, *queensAttacking*). AC and lRPC are better on the *geometric* class, though not by very large differences, at least compared to RNSAC and its variants. The only class where the singleton consistencies clearly fail is *queens*. This is because the constraint graph in *queens* is complete, meaning that NSAC is equivalent to SAC and the weaker methods are close to SAC. Also, the large size of domains entails a very large number of singleton checks. Among the classes missing from Table 1, some are trivial (e.g. *ehi*, *hanoi*, the rest of *composed*) and some are out of reach for all methods (*rlfap*, *black hole*, *haystacks*), which is partly due to the use of dom/ddeg. Class *pigeons* includes 3 instances that are not cut off. On these, lRPC is fastest, followed by RNSAC and its variants, being around 1.6 times slower, while all the rest are quite slower.

Among the singleton consistencies, it is clear that RNSAC is the best. It cuts down the search tree size almost as much as NSAC but it incurs considerably lower run times. The 1-pass variants are generally less efficient, but in some cases they offer advantages (*coloring* and *queensAttacking*). RsNSAC is usually less efficient than RNSAC, meaning that the extra propagation that the latter achieves pays off.

Figure 5 compares the cpu times between the best existing method (lRPC) and the best of the methods proposed here (RNSAC) including instances from all classes. It shows that there are many instances where lRPC is cut off while RNSAC

Table 1: Node visits (n), run times in secs (t), and number of timeouts (#TO) in summary. A (-) indicates that means could not be extracted due to the timeouts. After the name of each class we give the number of non-trivial instances excluding those where all methods timed out.

class	AC (n)	(t)	lRPC (n)	(t)	lmaxRPC (n)	(t)	NSAC (n)	(t)	NS1pAC (n)	(t)	RNSAC (n)	(t)	RNS1pAC (n)	(t)	RsNSAC (n)	(t)
qcp (15)																
mean	-	-	665,526	600	500,320	542	48,301	190	85,825	306	46,862	**134**	85,825	212	170,258	277
#TO	9		3		3		1		2		0		1		2	
qwh+ bqwh (120)																
mean	-	-	210,982	178	135,606	125	6,740	22	10,665	30	6,675	**15**	10,665	23	34,412	44
#TO	14		2		2		0		0		0		0		1	
graph coloring (22)																
mean	1,245,736	192	214,633	**66**	211,487	87	202,466	100	201,819	83	202,466	**65**	202,095	51	198,933	59
#TO	0		0		0		1		1		0		0		0	
comp-25-10 (10)																
mean	-	-	1,453,249	592	1,259,839	573	105	**0.30**	107	**0.29**	219	**0.23**	87,301	7	101,571	12
#TO	5		3		2		0		0		1		1		3	
geometric (100)																
mean	148,839	120	68,379	**112**	45,807	339	15,944	840	19,116	917	20,237	247	24,034	268	26,900	262
#TO	0		0		0		0		1		0		0		0	
queens (5)																
mean	4470	**15**	4470	30	4470	1529	-	-	-	-	-	-	-	-	-	-
#TO	0		1		2		4		4		4		4		4	
qKnights (18)																
mean	739K	647	739K	742	739K	953	0	**0.01**	0	**0.01**	0	**0.01**	0	**0.01**	0	**0.01**
#TO	12		12		12		0		0		0		0		0	
qAttacking (10)																
mean	-	-	-	-	-	-	19	236	19	148	23	68	23	**41**	26	60
#TO	3		3		3		1		1		0		0		0	

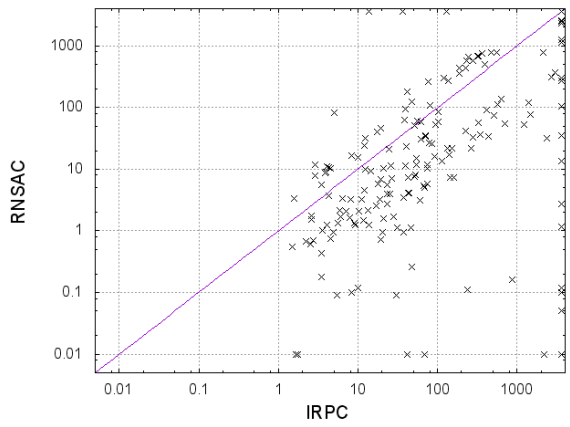

Figure 5: lRPC vs. RNSAC.

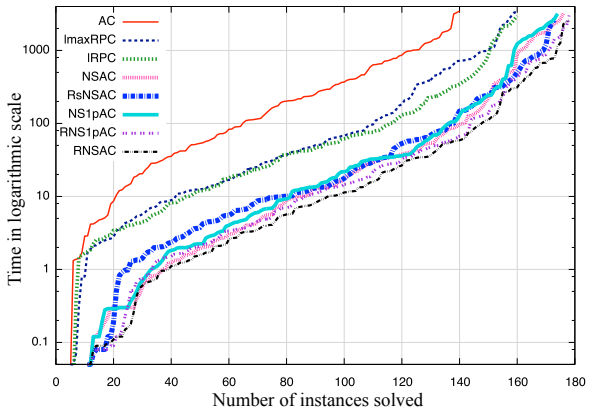

Figure 6: Number of instances solved per algorithm as the time allowed increases (cactus plot).

terminates. The opposite occurs only on instances of *queens*. In addition, there are numerous instances where exponential differences in favour of RNSAC occur.

Figure 6 displays a cactus plot giving the number of instances solved per algorithm as the time limit increases, excluding very hard, trivial, and very easy instances (leaving around 180 instances). We see that AC is competitive to lRPC and lmaxRPC only on instances that are easy, but its performance quickly starts to deteriorate. The two triangle based consistencies are closely matched, with lRPC being better on harder instances. Importantly, all singleton consistences solve more instances at any given time limit, with RNSAC being the best. RsNSAC is the worst on easy problems, while NSAC and NS1pAC are the less efficient on hard ones.

Finally, we have also experimented with the dom/wdeg heuristic. Results show that the neighborhood methods continue to perform very well on *quasigroup* problems as well as *queensKnights*/*queensAttacking*, are quite competitive on graph coloring, but are considerably slower on *rlfap* and *queens*. The composed class is trivial for dom/wdeg while *black hole* and *haystacks* remain very hard.

7 Conclusion

Using NSAC as basis, we have proposed and studied, both theoretically and experimentally, a family of singleton consistencies that are inspired by either maxRPC or RPC, and are very easy to implement. Theoretical results show that the pruning power of these consistencies lays between RPC and NSAC. Experimental results show that these local consistencies, and particularly RNSAC, display very good performance. It is important to explore the viability of neighborhood singleton consistencies for the case of non-binary constraints. Strong local consistency methods for non-binary constraints do exist for some classes of constraints (most notably table constraints) but they are typically implemented through algorithms or reformulations that are complex or/and have high space requirements. Singleton neighborhood methods offer the very interesting possibility of easily achieving strong pruning using existing, highly efficient, algorithms for GAC or even lesser levels of consistency.

References

[Balafoutis *et al.*, 2011] Thanasis Balafoutis, Anastasia Paparrizou, Kostas Stergiou, and Toby Walsh. New algorithms for max restricted path consistency. *Constraints*, 16(4):372–406, 2011.

[Balafrej *et al.*, 2014] Amine Balafrej, Christian Bessiere, El-Houssine Bouyakh, and Gilles Trombettoni. Adaptive Singleton-Based Consistencies. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pages 2601–2607, 2014.

[Berlandier, 1995] Pierre Berlandier. Improving Domain Filtering Using Restricted Path Consistency. In *Proceedings of IEEE Conference on Artificial Intelligence and Applications (CAIA'95)*, pages 32–37, 1995.

[Bessiere *et al.*, 2011] Christian Bessiere, Stéphane Cardon, Romuald Debruyne, and Christophe Lecoutre. Efficient Algorithms for Singleton Arc Consistency. *Constraints*, 16:25–53, 2011.

[Debruyne and Bessiere, 1997] Romuald Debruyne and Christian Bessiere. From restricted path consistency to max-restricted path consistency. In *Proceedings of 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 312–326, 1997.

[Debruyne and Bessiere, 2001] Romuald Debruyne and Christian Bessiere. Domain Filtering Consistencies. *J. Artif. Intell. Res. (JAIR)*, 14:205–230, 2001.

[Freuder and Elfe, 1996] Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of the 13th AAAI Conference on Artificial Intelligence (AAAI'96)*, pages 202–208, 1996.

[Lecoutre and Hemery, 2007] Christophe Lecoutre and Fred Hemery. A study of residual supports in arc consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 125–130, 2007.

[Likitvivatanavong *et al.*, 2007] Chavalit Likitvivatanavong, Yuanlin Zhang, James Bowen, Scott Shannon, and Eugene C. Freuder. Arc Consistency during Search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 137–142, 2007.

[Stergiou, 2015] Kostas Stergiou. Restricted Path Consistency Revisited. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP'15)*, pages 419–428, 2015.

[Vion and Debruyne, 2009] Julien Vion and Romuald Debruyne. Light Algorithms for Maintaining Max-RPC During Search. In *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, pages 167–174, 2009.

[Wallace, 2015] Richard J. Wallace. SAC and neighbourhood SAC. *AI Communications*, 28(2):345–364, 2015.

[Wallace, 2016a] Richard J. Wallace. Neighbourhood SAC: Extensions and new algorithms. *AI Communications*, 29(2):249–268, 2016.

[Wallace, 2016b] Richard J. Wallace. Preprocessing versus search processing for constraint satisfaction problems. In *Proceedings of Knowledge Representation and Automated Reasoning (RCRA'16)*, pages 89–103, 2016.