Adaptive Singleton-based Consistencies*

Amine Balafrej
CNRS, U. Montpellier, France
U. Mohammed V Agdal, Morocco

Christian Bessiere
CNRS, U. Montpellier
France

El Houssine Bouyakhf
FSR, U. Mohammed V Agdal
Morocco

Gilles Trombettoni
CNRS, U. Montpellier
France

Abstract

Singleton-based consistencies have been shown to dramatically improve the performance of constraint solvers on some difficult instances. However, they are in general too expensive to be applied exhaustively during the whole search. In this paper, we focus on partition-one-AC, a singleton-based consistency which, as opposed to singleton arc consistency, is able to prune values on all variables when it performs singleton tests on one of them. We propose adaptive variants of partition-one-AC that do not necessarily run until having proved the fixpoint. The pruning can be weaker than the full version but the computational effort can be significantly reduced. Our experiments show that adaptive Partition-one-AC can obtain significant speed-ups over arc consistency and over the full version of partition-one-AC.

1 Introduction

The standard level of consistency that constraint solvers apply during search for solutions in constraint networks is arc consistency (AC). Applying stronger levels of consistency can improve standard AC on some difficult instances, but they are in general too expensive to be applied exhaustively everywhere on the network during the whole search.

Some recent works apply a level of strong consistency in a non exhaustive way. In (Stamatatos and Stergiou 2009), a preprocessing phase learns which level of consistency to apply on which parts of the instance. Once the level is learned, it is statically applied during the whole search. In (Stergiou 2008; Paparrizou and Stergiou 2012), some heuristics allow the solver to dynamically select during search AC or a stronger level of consistency (maxRPC) depending on the variable/constraint. In (Balafrej et al. 2013), for each variable/constraint the solver learns during search a parameter that characterizes a parameterized level of consistency to apply on the variable/constraint. That parameterized level lies between AC and a stronger level.

In this paper we focus on singleton-based consistencies. They have been shown extremely efficient to solve some classes of hard problems (Bessiere et al. 2011). Singleton-based consistencies apply the *singleton test* principle, which consists in assigning a value to a variable and trying to refute it by enforcing a given level of consistency. If a contradiction occurs during this singleton test, the value is removed from its domain. The first example of such a local consistency is Singleton Arc Consistency (SAC), introduced in (Debruyne and Bessiere 1997). In SAC, the singleton test enforces arc consistency. By definition, SAC can only prune values in the variable domain on which it currently performs singleton tests. In (Bennaceur and Affane 2001), Partition-One-AC (that we call POAC) has been proposed. POAC is an extension of SAC that can prune values everywhere in the network as soon as a variable has been completely singleton tested. As a consequence, the fixpoint in terms of filtering is often quickly reached in practice. This observation has already been made on numerical CSPs. In (Trombettoni and Chabert 2007; Neveu and Trombettoni 2013) a consistency called Constructive Interval Disjunction (CID), close to POAC in its principle, gave good results by simply calling the main procedure once on each variable or by adapting during search the number of times it is called.

Based on these observations, we propose an adaptive version of POAC, called APOAC, where the number of times variables are processed for singleton tests on their values is dynamically and automatically adapted during search. A sequence of singleton tests on all values of one variable is called a `varPOAC` call. The number k of times `varPOAC` is called will depend on how much POAC is efficient or not in pruning values. This number k of `varPOAC` calls will be learned during a sequence of nodes of the search tree (learning nodes) by measuring a stagnation in the amount of pruned values. This amount k of `varPOAC` calls will be applied at each node during a sequence of nodes (called exploitation nodes) before we enter a new learning phase to adapt k again. Observe that if the number of `varPOAC` calls learned is 0, the adaptive POAC will mimic AC.

The rest of the paper is organized as follows. Section 2 contains the necessary formal background. In Section 3, we propose an efficient POAC algorithm that will be used as a basis for our adaptive versions. We give a quick comparison to SAC that validates our choices. Section 4 presents our different approaches to learn the number of `varPOAC` calls to apply. In Section 5, hard instances from the 2009 CSP competition are used to experimentally select the best among our learning mechanisms and to validate our approach as a way

*Supported by the EU project ICON (FP7-284715).

to improve the performance of standard AC-based solvers on instances difficult to solve. Section 6 concludes this work.

2 Background

A *constraint network* is defined as a set of n variables $X = \{x_1, ..., x_n\}$, a set of domains $D = \{D(x_1), ..., D(x_n)\}$, and a set of e constraints $C = \{c_1, ..., c_e\}$. Each constraint c_k is defined by a pair $(var(c_k), sol(c_k))$, where $var(c_k)$ is an ordered subset of X, and $sol(c_k)$ is a set of combinations of values (tuples) satisfying c_k. A tuple $t \in sol(c_k)$ is valid iff $\forall x_i \in var(c_k)$, $t[x_i] \in D(x_i)$, where $t[x_i]$ is the value that x_i takes in t. In the following $\Gamma(x_i)$ will denote the set of constraints c_j such that $x_i \in var(c_j)$.

A tuple $t \in sol(c_k)$ is called a *support* for $v_i \in D(x_i)$, $x_i \in var(c_k)$, on c_k iff t is valid and $t[x_i] = v_i$. A value $v_i \in D(x_i)$ is *arc consistent* iff for all $c_j \in \Gamma(x_i)$ v_i has a support on c_j. A variable x_i is arc consistent if $D(x_i) \neq \emptyset$ and all values in $D(x_i)$ are arc consistent. A network is arc consistent if all its variables are arc consistent. We denote by $AC(N)$ the network obtained by enforcing arc consistency on N. If $AC(N)$ has empty domains, we say that N is arc inconsistent.

Given a constraint network $N = (X, D, C)$, a value $v_i \in D(x_i)$ is *singleton arc consistent* (SAC) iff the network $N|_{x_i=v_i}$ where $D(x_i)$ is reduced to the singleton $\{v_i\}$ is not arc inconsistent. A variable x_i is SAC if $D(x_i) \neq \emptyset$ and all values in $D(x_i)$ are SAC. A network is SAC if all its variables are SAC.

Given a constraint network $N = (X, D, C)$, a variable x_i is partition-one-AC (POAC) iff $D(x_i) \neq \emptyset$, all values in $D(x_i)$ are SAC, and $\forall j \in 1..n, j \neq i, \forall v_j \in D(x_j)$, $\exists v_i \in D(x_i)$ such that $v_j \in AC(N|_{x_i=v_i})$. A constraint network $N = (X, D, C)$ is POAC iff all its variables are POAC. Observe that POAC, as opposed to SAC, is able to prune values from all variable domains when being enforced on a given variable.

It has been shown in (Bennaceur and Affane 2001) that POAC is strictly stronger than SAC. Following (Debruyne and Bessiere 1997), this means that SAC holds on any constraint network on which POAC holds and there exists a constraint network on which SAC holds but not POAC.

3 POAC

Before moving to adaptive partition-one-AC, we first propose an efficient algorithm enforcing POAC and we compare its behaviour to SAC.

3.1 The Algorithm

The efficiency of our POAC algorithm, `POAC1`, is based on the use of counters associated with each value in the constraint network. These counters are used to count how many times a value v_j from a variable x_j is pruned during the sequence of POAC tests on all the values of another variable x_i (the `varPOAC` call to x_i). If v_j is pruned $|D(x_i)|$ times, this means that it is not POAC and can be removed from $D(x_j)$.

`POAC1` (Algorithm 1) starts by enforcing arc consistency on the network (line 2). Then it puts all variables in the ordered cyclic list S using any total ordering on X (line 4).

Algorithm 1: `POAC1`(X, D, C)

```
1  begin
2      if ¬EnforceAC$(X, D, C)$ then
3          return $false$
4      $S \leftarrow CyclicList(Ordering(X))$
5      FPP $\leftarrow 0$
6      $x_i \leftarrow first(S)$
7      while FPP $< |X|$ do
8          if ¬varPOAC$(x_i, X, D, C,$ CHANGE$)$ then
9              return $false$
10         if CHANGE then FPP $\leftarrow 1$ else FPP++
11         $x_i \leftarrow NextElement(x_i, S)$
12     return $true$
```

`varPOAC` iterates on all variables from S (line 8) to make them POAC until the fixpoint is reached (line 12) or a domain wipe-out occurs (line 9). The counter FPP (FixPoint Proof) counts how many calls to `varPOAC` have been processed in a row without any change in any domain (line 10).

The procedure `varPOAC` (Algorithm 2) is called to establish POAC w.r.t. a variable x_i. It works in two steps. The first step enforces arc consistency in each sub-network $N = (X, D, C \cup \{x_i = v_i\})$ (line 4) and removes v_i from $D(x_i)$ (line 5) if the sub-network is arc-inconsistent. Otherwise, the procedure `TestAC` (Algorithm 3) increments the counter associated with every arc inconsistent value $(x_j, v_j), j \neq i$ in the sub-network $N = (X, D, C \cup \{x_i = v_i\})$. (Lines 6 and 7 have been added for improving the performance in practice but are not necessary for reaching the required level of consistency.) In line 9 the Boolean CHANGE is set to $true$ if $D(x_i)$ has changed. The second step deletes all the values $(x_j, v_j), j \neq i$ with a counter equal to $|D(x_i)|$ and sets back the counter of each value to 0 (lines 13-15). Whenever a domain change occurs in $D(x_j)$, if the domain is empty, `varPOAC` returns failure (line 16); otherwise it sets the Boolean CHANGE to $true$ (line 17).

Enforcing arc consistency on the sub-networks $N = (X, D, C \cup \{x_i = v_i\})$ is done by calling the procedure `TestAC` (Algorithm 3). `TestAC` just checks whether arc consistency on the sub-network $N = (X, D, C \cup \{x_i = v_i\})$ leads to a domain wipe-out or not. It is an instrumented AC algorithm that increments a counter for all removed values and restores them all at the end. In addition to the standard propagation queue Q, `TestAC` uses a list L to store all the removed values. After the initialisation of Q and L (lines 2-3), `TestAC` revises each arc (x_j, c_k) in Q and adds each removed value (x_j, v_j) to L (lines 5-10). If a domain wipe-out occurs (line 11), `TestAC` restores all removed values (line 12) without incrementing the counters (call to `RestoreDomains` with UPDATE $= false$) and it returns failure (line 13). Otherwise, if values have been pruned from the revised variable (line 14) it puts in Q the neighbour arcs to be revised. At the end, removed values are restored (line 16) and their counters are incremented (call to `RestoreDomains` with UPDATE $= true$) before returning success (line 17).

Algorithm 2: `varPOAC`(x_i, X, D, C, CHANGE)

```
begin
    SIZE ← $|D(x_i)|$; CHANGE ← $false$
    foreach $v_i \in D(x_i)$ do
        if ¬TestAC($X, D, C \cup \{x_i = v_i\}$) then
            remove $v_i$ from $D(x_i)$
            if ¬EnforceAC($X, D, C, x_i$) then
                return $false$
    if $D(x_i) = \emptyset$ then return $false$
    if SIZE $\neq |D(x_i)|$ then CHANGE ← $true$
    foreach $x_j \in X \backslash \{x_i\}$ do
        SIZE ← $|D(x_j)|$
        foreach $v_j \in D(x_j)$ do
            if counter($x_j, v_j$) $= |D(x_i)|$ then
                remove $v_j$ from $D(x_j)$
            counter($x_j, v_j$) ← 0
        if $D(x_j) = \emptyset$ then return $false$
        if SIZE $\neq |D(x_j)|$ then CHANGE ← $true$
    return $true$
```

Proposition 1 `POAC1` *has a* $O(n^2d^2(T + n))$ *worst-case time complexity, where* T *is the time complexity of the arc-consistency algorithm used for singleton tests,* n *is the number of variables, and* d *is the number of values in the largest domain.*

Proof. The cost of calling `varPOAC` on a single variable is $O(dT + nd)$ because `varPOAC` runs AC on d values and updates nd counters. In the worst case, each of the nd value removals provoke n calls to `varPOAC`. Therefore `POAC1` has a time complexity in $O(n^2d^2(T + n))$. □

3.2 Comparison of POAC and SAC behaviors

Although POAC has a worst-case time complexity greater than SAC, we observed in practice that maintaining POAC during search is often faster than maintaining SAC. This behavior occurs even when POAC cannot remove more values than SAC, i.e., when the same number of nodes is visited with the same static variable ordering. This is due to what we call the *(filtering) convergence speed*: when both POAC and SAC reach the same fixpoint, POAC reaches the fixpoint with less singleton tests than SAC.

Figure 1 compares the convergence speed of POAC and SAC on an instance where they have the same fixpoint. We observe that POAC is able to reduce the domains, to reach the fixpoint, and to prove the fixpoint, all in less singleton tests than SAC. This pattern has been observed on most of the instances and whatever ordering was used in the list S. The reason is that each time POAC applies `varPOAC` to a variable x_i, it is able to remove inconsistent values from $D(x_i)$ (like SAC), but also from any other variable domain (unlike SAC).

The fact that SAC cannot remove values in variables other than the one on which the singleton test is performed makes it a poor candidate for adapting the number of singleton tests. A SAC-inconsistent variable/value pair never singleton tested has no chance to be pruned by such a technique.

Algorithm 3: `TestAC`($X, D, C \cup \{x_i = v_i\}$)

```
begin
    $Q \leftarrow \{(x_j, c_k) | c_k \in \Gamma(x_i), x_j \in var(c_k), x_j \neq x_i\}$
    $L \leftarrow \emptyset$
    while $Q \neq \emptyset$ do
        pick and delete $(x_j, c_k)$ from $Q$
        SIZE ← $|D(x_j)|$
        foreach $v_j \in D(x_j)$ do
            if ¬HasSupport($x_j, v_j, c_k$) then
                remove $v_j$ from $D(x_j)$
                $L \leftarrow L \cup (x_j, v_j)$
        if $D(x_j) = \emptyset$ then
            RestoreDomains($L, false$)
            return $false$
        if $|D(x_j)| <$ SIZE then
            $Q \leftarrow Q \cup \{(x_{j'}, c_{k'}) | c_{k'} \in \Gamma(x_j), x_{j'} \in$
            $var(c_{k'}), x_{j'} \neq x_j, c_{k'} \neq c_k\}$
    RestoreDomains($L, true$)
    return $true$
```

Algorithm 4: `RestoreDomains`(L, UPDATE)

```
begin
    if UPDATE then
        foreach $(x_j, v_j) \in L$ do
            $D(x_j) \leftarrow D(x_j) \cup \{v_j\}$
            counter($x_j, v_j$) ← counter($x_j, v_j$) + 1
    else
        foreach $(x_j, v_j) \in L$ do
            $D(x_j) \leftarrow D(x_j) \cup \{v_j\}$
```

4 Adaptive POAC

This section presents an adaptive version of POAC that approximates POAC by monitoring the number of variables on which to perform singleton tests.

To achieve POAC, `POAC1` calls the procedure `varPOAC` until it has proved that the fixpoint is reached. This means that, when the fixpoint is reached, `POAC1` needs to call n (additional) times the procedure `varPOAC` without any pruning to prove that the fixpoint was reached. Furthermore, we experimentally observed that in most cases there is a long sequence of calls to `varPOAC` that prune very few values, even before the fixpoint has been reached (see Figure 1 as an example). The goal of *Adaptive POAC* (APOAC) is to stop iterating on `varPOAC` as soon as possible. We want to benefit from strong propagation of singleton tests while avoiding the cost of the last calls to `varPOAC` that delete very few values or no value at all.

4.1 Principle

The APOAC approach alternates between two phases during search: a short *learning* phase and a longer *exploitation*

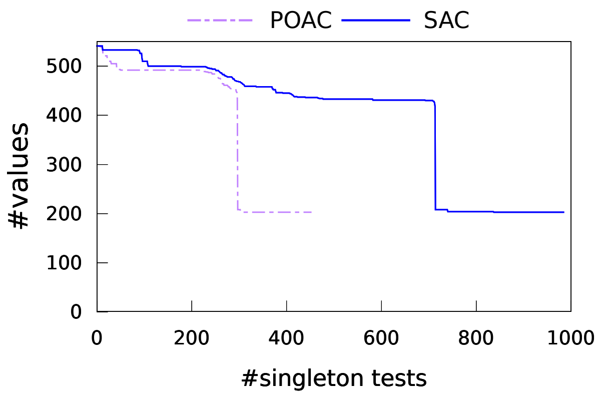

Figure 1: The convergence speed of POAC and SAC.

phase. One of the two phases is executed on a sequence of nodes before switching to the other phase for another sequence of nodes. The search starts with a learning phase. The total length of a pair of sequences learning + exploitation is fixed to the parameter LE.

Before going to a more detailed description, let us define the (log_2 of the) *volume* of a constraint network $N = (X, D, C)$, used to approximate the size of the search space:

$$V = log_2 \prod_{i=1}^{n} |D(x_i)|$$

We use the logarithm of the volume instead of the volume itself because of the large integers the volume generates. We also could have used the perimeter (i.e., $\sum_i |D(x_i)|$) for approximating the search space, as done in (Neveu and Trombettoni 2013). However, experiments have confirmed that the volume is a finer and better criterion for adaptive POAC.

The ith learning phase is applied to a sequence of $L = \frac{1}{10} \cdot LE$ consecutive nodes. During that phase, we learn a cutoff value k_i which is the maximum number of calls to the procedure `varPOAC` that each node of the next (ith) exploitation phase will be allowed to perform. A good cutoff k_i is such that `varPOAC` removes many inconsistent values (that is, obtains a significant volume reduction in the network) while avoiding calls to `varPOAC` that delete very few values or no value at all. During the ith exploitation phase, applied to a sequence of $\frac{9}{10} \cdot LE$ consecutive nodes, the procedure `varPOAC` is called at each node until fixpoint is proved or the cutoff limit of k_i calls to `varPOAC` is reached.

The ith learning phase works as follows. Let k_{i-1} be the cutoff learned at the previous learning phase. We initialize $maxK$ to $max(2 \cdot k_{i-1}, 2)$. At each node n_j in the new learning sequence $n_1, n_2, \ldots n_L$, APOAC is used with a cutoff $maxK$ on the number of calls to the procedure `varPOAC`. APOAC stores the sequence of volumes $(V_1, \ldots, V_{last})$, where V_p is the volume resulting from the pth call to `varPOAC` and $last$ is the smallest among $maxK$ and the number of calls needed to prove fixpoint. Once the fixpoint is proved or the $maxK$th call to `varPOAC` performed, APOAC computes $k_i(j)$, the number of `varPOAC` calls that are enough to *sufficiently* reduce the volume while avoiding the extra cost of the last calls that remove few or no value. (The criteria to decide what 'sufficiently' means are described in Section 4.2.) Then, to make the learning phase more adaptive, $maxK$ is updated before starting node n_{j+1}. If $k_i(j)$ is close to $maxK$, that is, greater than $\frac{3}{4}maxK$, we increase $maxK$ by 20%. If $k_i(j)$ is less than $\frac{1}{2}maxK$, we reduce $maxK$ by 20%. Otherwise $maxK$ is unchanged. Once the learning phase ends, APOAC computes the cutoff k_i that will be applied to the next exploitation phase. k_i is an aggregation of the $k_i(j)$s, $j = 1, ..., L$, following one of the aggregation techniques presented in Section 4.3.

4.2 Computing $k_i(j)$

We implemented APOAC using two different techniques to compute $k_i(j)$ at a node n_j of the learning phase:

- LR (*Last Reduction*) $k_i(j)$ is the rank of the last call to `varPOAC` that reduced the volume of the constraint network.
- LD (*Last Drop*) $k_i(j)$ is the rank of the last call to `varPOAC` that has produced a *significant* drop of the volume. The significance of a drop is captured by a ratio $\beta \in [0, 1]$. More formally, $k_i(j) = max\{p \mid V_p \leq (1-\beta)V_{p-1}\}$.

4.3 Aggregation of the $k_i(j)$s

Once the ith learning phase completed, APOAC aggregates the $k_i(j)$s computed during that phase to generate k_i, the new cutoff value on the number of calls to the procedure `varPOAC` allowed at each node of the ith exploitation phase. We propose two techniques to aggregate the $k_i(j)$s into k_i.

- Med k_i is the median of the $k_i(j), j \in 1..L$.
- q-PER This technique generalizes the previous one. Instead of taking the median, we use any percentile. That is, k_i is equal to the smallest value among $k_i(1), \ldots, k_i(L)$ such that q% of the values among $k_i(1), \ldots, k_i(L)$ are less than or equal to k_i.

Several variants of APOAC can be proposed, depending on how we compute the $k_i(j)$ values in the learning phase and how we aggregate the different $k_i(j)$s. In the next section, we give an experimental comparison of the different variants we tested.

5 Experiments

This section presents experiments that compare the performance of maintaining AC, POAC, or adaptive variants of POAC during search. For the adaptive variants we use two techniques to determine $k_i(j)$: the last reduction (LR) and the last drop (LD) with $\beta = 5\%$ (see Section 4.2). We also use two techniques to aggregate these $k_i(j)$s: the median (Med) and the qth percentile (q-PER) with $q = 70\%$ (see Section 4.3). In experiments not presented in this paper we tested the performance of APOAC using percentiles $10, 20, \ldots 90$. The 70th percentile showed the best behavior. We have performed experiments for the four variants obtained by combining two by two the parameters LR vs LD

Table 1: Total number of instances solved by AC, several variants of APOAC, and POAC.

$k_i(j)$	k_i		**AC**	**APOAC-2**	**APOAC-n**	**APOAC-fp**	**POAC**
LR	70-PER	**#solved**	115	116	**119**	118	115
	Med	**#solved**	115	114	**118**	**118**	115
LD	70-PER	**#solved**	115	117	**121**	120	115
	Med	**#solved**	115	116	**119**	**119**	115

Table 2: CPU time for AC, APOAC-2, APOAC-n, APOAC-fp and POAC on the eight problem classes.

	#SolvedbyOne		**AC**	**APOAC-2**	**APOAC-n**	**APOAC-fp**	**POAC**
Tsp-20	**15(/15)**	**#solved instances**	**15**	15	15	15	15
		sum CPU(s)	**1,596.38**	3,215.07	4,830.10	7,768.33	18,878.81
Tsp-25	**15(/15)**	**#solved instances**	15	14	**15**	15	11
		sum CPU(s)	20,260.08	>37,160.63	**16,408.35**	33,546.10	>100,947.01
renault	**50(/50)**	**#solved instances**	**50**	50	50	50	50
		sum CPU(s)	**837.72**	2,885.66	11,488.61	15,673.81	18,660.01
cril	**7(/8)**	**#solved instances**	4	5	**7**	7	7
		sum CPU(s)	>45,332.55	>42,436.17	**747.05**	876.57	1,882.88
mug	**6(/8)**	**#solved instances**	5	6	6	6	**6**
		sum CPU(s)	>29,931.45	12,267.39	12,491.38	12,475.66	**2,758.10**
K-insertions	**6(/10)**	**#solved instances**	4	5	**6**	5	5
		sum CPU(s)	>30,614.45	>29,229.71	**27,775.40**	>29,839.39	>20,790.69
myciel	**12(/15)**	**#solved instances**	**12**	12	12	12	11
		sum CPU(s)	**1,737.12**	2,490.15	2,688.80	2,695.32	>20,399.70
Qwh-20	**10(/10)**	**#solved instances**	10	10	**10**	10	10
		sum CPU(s)	16,489.63	12,588.54	**11,791.27**	12,333.89	27,033.73
	Sum of CPU times		*>146,799*	*>142,273*	***88,221***	*>115,209*	*>211,351*
Sum of average CPU times per class			*>18,484*	*>14,717*	***8,773***	*>9,467*	*>10,229*

and Med vs 70-PER. For each variant we compared three initial values for the initial $maxK$ used by the first learning phase: $maxK \in \{2, n, \infty\}$, where n is the number of variable in the instance to be solved. These three versions are denoted respectively by APOAC-2, APOAC-n and APOAC-fp.

We compare these search algorithms on problems available from Lecoutre's webpage.[1] We selected four binary classes containing at least one difficult instance for MAC ($>$ 10 seconds): mug, K-insertions, myciel and Qwh-20. We also selected all the n-ary classes in extension: the traveling-salesman problem (TSP-20, TSP-25), the Renault Megane configuration problem (Renault) and the Cril instances (Cril). These eight problem classes contain instances with 11 to 1406 variables, domains of size 3 to 1600 and 20 to 9695 constraints.

For the search algorithm maintaining AC, the algorithm AC2001 (resp. GAC2001) (Bessiere et al. 2005) is used for the binary (resp. non-binary) problems. The same AC algorithms are used as refutation procedure for POAC and APOAC algorithms. The $dom/wdeg$ heuristic (Boussemart et al. 2004) is used both to order variables in the $Ordering(X)$ function (see line 4 of Algorithm 1) and to order variables during search for all the search algorithms. The results presented involve all the instances solved before the cutoff of 15,000 seconds by at least one algorithm. All the algorithms are implemented in our JAVA CSP solver.

Table 1 compares all the competitors and gives the number of instances (#solved) solved before the cutoff. We observe that, on the set of instances tested, adaptive versions of POAC are better than AC and POAC. All of them, except APOAC-2+LR+Med, solve more instances than AC and POAC. All the versions using the last drop (LD) technique to determine the $k_i(j)$s in the learning phase are better than those using the last reduction (LR). We also see that the versions that use the 70th percentile (70-PER) to aggregate the $k_i(j)$s are better than those using the median (Med). This suggests that the best combination is LD+70-PER. This is the only combination we will use in the following.

Table 2 focuses on the performance of the three variants of APOAC (APOAC-2, APOAC-n and APOAC-fp), all with the combination (LD+70-PER). The second column reports the number #SolvedbyOne of instances solved before the cutoff by at least one algorithm. For each algorithm and each class, Table 2 shows the sum of CPU times required to solve those #SolvedbyOne instances. When a competitor cannot solve an instance before the cutoff, we count 15,000 seconds for that instance and we write '>' in front of the corresponding sum of CPU times. The last two rows of the table give the total sum of CPU times and the sum of average CPU times per class. For each class taken separately,

[1]www.cril.univ-artois.fr/~lecoutre/benchmarks.html

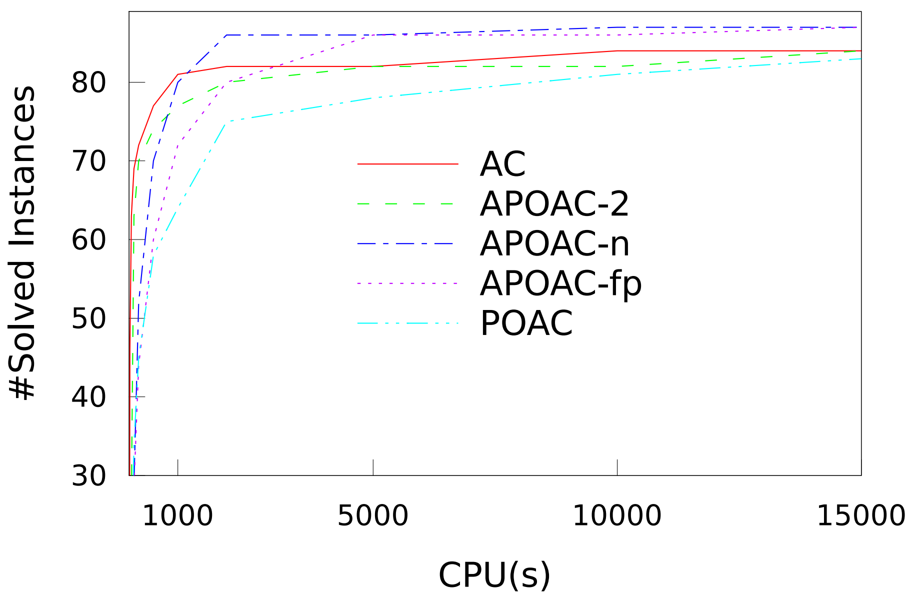

Figure 2: Number of instances solved when the time allowed increases.

the three versions of APOAC are never worse than AC and POAC at the same time. APOAC-n solves all the instances solved by AC and POAC and for four of the eight problem classes it outperforms both AC and POAC. However, there remain a few classes, such as Tsp-20 and renault, where even the first learning phase of APOAC is too costly to compete with AC despite our agile auto-adaptation policy that limits the number of calls to `varPOAC` during learning (see Section 4.1). Table 2 also shows that maintaining a high level of consistency such as POAC throughout the whole network generally produces a significant overhead.

Table 3 and Figure 2 sum up the performance results obtained on all the instances with n-ary constraints. The binary classes are not taken into account by these summarized table and figure because they have not been exhaustively tested. Figure 2 gives the performance profile for each algorithm presented in Table 2: AC, APOAC-2, APOAC-n, APOAC-fp and POAC. Each point (t, i) on a curve indicates the number i of instances that an algorithm can solve in less than t seconds. The performance profile underlines that AC and APOAC are better than POAC: whatever the time given, they solve more instances than POAC. The comparison between AC and APOAC highlights two phases. A first phase (for easy instances) where AC is better than APOAC, and a second phase where APOAC becomes better than AC. Among the adaptive versions, APOAC-n is the variant with the shortest first phase (it adapts quite well to easy instances) and it remains the best even when time increases.

Finally, Table 3 compares the best APOAC version (APOAC-n) to AC and POAC on n-ary problems. The first row of the table gives the number of solved instances by each algorithm before the cutoff. We observe that APOAC-n solves more instances than AC and POAC. The second row of the table gives the sum of CPU time required to solve all the instances. Again, when an instance cannot be solved before the cutoff of 15,000 seconds, we count 15,000 seconds for that instance. We observe that APOAC-n significantly outperforms both AC and POAC. The last two rows of the table give the gain of APOAC-n w.r.t. AC and w.r.t. POAC. We see that APOAC-n has a positive total gain greater than 51% compared to AC and greater than 76% compared to POAC.

Table 3: Performance of APOAC-n compared to AC and POAC on n-ary problems.

	AC	**APOAC-n**	**POAC**
#solved instances	84(/87)	**87(/87)**	83(/87)
sum CPU(s)	>68,027	**33,474**	>140,369
gain w.r.t. AC	–	*>51%*	–
gain w.r.t. POAC	–	*>76%*	–

6 Conclusion

We have proposed `POAC1`, an algorithm that enforces partition-one-AC efficiently in practice. We have shown that POAC converges faster than SAC to the fixpoint due to its ability to prune values from all variable domains when being enforced on a given variable. We proposed an adaptive version of POAC that monitors the number of variables on which to perform singleton tests. Experiments show that the adaptive version of POAC gives a good trade-off between filtering and search. This leads to a number of solved instances greater than with AC or POAC and in a significant gain in CPU time.

References

Balafrej, A.; Bessiere, C.; Coletta, R.; and Bouyakhf, E. 2013. Adaptive parameterized consistency. In *Proceedings of the Eighteenth International Conference on Principles and Practice of Constraint Programming (CP'13), LNCS 8124, Springer–Verlag*, 143–158.

Bennaceur, H., and Affane, M.-S. 2001. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP'01), LNCS 2239, Springer–Verlag*, 560–564.

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

Bessiere, C.; Cardon, S.; Debruyne, R.; and Lecoutre, C. 2011. Efficient algorithms for singleton arc consistency. *Constraints* 16(1):25–53.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI'04)*, 146–150.

Debruyne, R., and Bessiere, C. 1997. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, 412–417.

Neveu, B., and Trombettoni, G. 2013. Adaptive Constructive Interval Disjunction. In *Proceedings of the 25th IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'13)*, 900–906.

Paparrizou, A., and Stergiou, K. 2012. Evaluating simple fully automated heuristics for adaptive constraint propagation. In *Proceedings of the 24th IEEE International Conference on Tools for Artificial Intelligence (IEEE-ICTAI'12)*, 880–885.

Stamatatos, E., and Stergiou, K. 2009. Learning how to propagate using random probing. In *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09), LNCS 5547, Springer*, 263–278.

Stergiou, K. 2008. Heuristics for dynamically adapting propagation. In *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, 485–489.

Trombettoni, G., and Chabert, G. 2007. Constructive Interval Disjunction. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP'07), LNCS 4741, Springer–Verlag*, 635–650.