Cycle-Based Singleton Local Consistencies

Robert J. Woodward[1,2] Berthe Y. Choueiry[1]
Christian Bessiere[2]

[1]Constraint Systems Laboratory
University of Nebraska-Lincoln, USA
{rwoodwar|choueiry}@cse.unl.edu

[2]LIRMM-CNRS
University of Montpellier, France
bessiere@lirmm.fr

UNL-CSE-2016-0004

September 22, 2016

Abstract

We propose to exploit cycles in the constraint network of a Constraint Satisfaction Problem (CSP) to vehicle constraint propagation and improve the effectiveness of local consistency algorithms. We focus our attention on the consistency property Partition-One Arc-Consistency (POAC), which is a stronger variant of Singleton Arc-Consistency (SAC). We introduce rPOAC, a relational variant of POAC, defined on the relations of the CSP. We modify the algorithm for enforcing POAC and the algorithm's adaptive version to operate on a minimum cycle basis (MCB) of the incidence graph of the CSP for POAC and the CSP's minimal dual graph for rPOAC. We empirically show that our approach improves the performance of problem solving and constitutes a novel and effective localization of consistency algorithms.

1 Introduction

While Singleton Arc-Consistency (SAC) is a strong form of consistency [Debruyne and Bessière, 1997], the cost of enforcing it on a Constraint Satisfaction Problem (CSP) during search is prohibitive in practice. The algorithm for SAC operates by assigning a value to a variable then enforcing arc consistency (AC) on the entire CSP. That operation is called the *singleton test*. If the problem is found to be inconsistent, the value is removed from the domain of the tested variable. Wallace [2015] proposed NSAC, which reduces the cost of enforcing SAC by restricting AC to the neighborhood of the variable. Bennaceur and Affane [2001] proposed Partition-One Arc-Consistency (POAC), an extension of SAC, to prune values *anywhere* in the problem as soon as a variable has been completely singleton tested. POAC can *both* reduce the cost of SAC and increase its filtering. Balafrej *et al.* [2014] showed that the cost of POAC, when used for real-full lookahead (RFL) during search, can be further reduced by interrupting it before a fixed point is reached. They proposed APOAC as an adaptive version of POAC, where the "number of times variables are processed for singleton tests on their values is dynamically and automatically adapted during search."

In this paper, we explore using localization to improve the performance of POAC. As a first step, we replicate the approach of Wallace [2015] for SAC and restrict POAC to the direct neighborhood of each variable. Then, we explore a completely new direction: we restrict POAC to cycles in which a variable participates, the rationale being that cycles are likely to expose inconsistencies. We explore the use of a Minimum Cycle Basis (MCB) of a variable, which can be efficiently computed [Horton, 1987], generating MCBs on the incidence graph and a minimal dual graph of the CSP. Finally, we empirically validate our approach.

Although this paper focuses on POAC, we believe that exploiting cycles, such as MCBs, is applicable to other consistency algorithms, and that our study opens a new direction in the design of consistency algorithms.

This paper is structured as follows. Section 2 reviews background information about CSPs. Section 3 introduces our localized variant of POAC that uses a minimum cycle basis to define the subproblem over which POAC singleton tests. Section 4 empirically compares the performance of algorithms for POAC. Finally, Section 5 concludes this paper.

2 Background

Below, we review the basic concepts used in this paper.

2.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined by $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X}$ is a set of variables, and each variable $x_i \in \mathcal{X}$ has a finite domain $dom(x_i) \in \mathcal{D}$. Let (x_i, v_i) denote a variable-value pair, $(x_i, v_i) \in \mathcal{P}$ if $v_i \in dom(x_i)$. Each constraint $c_i \in \mathcal{C}$ is specified by its scope, $scope(c_i)$, which is the set of variables to which c_i applies, and by a relation R_i, which is a set of allowed tuples ($R_i \subseteq \prod_{x_i \in scope(c_i)} dom(x_i)$). We denote by $neigh(c_i)$ the set of constraints whose scopes intersect with that of c_i, $cons(x_i)$ the set of constraints that apply to x_i, and $neigh(x_i) = \cup_{c \in cons(x_i)} scope(c) \setminus \{x_i\}$. A binary CSP is a CSP whose constraints are over two variables. A solution to the CSP assigns, to each variable, a value taken from its domain such that all the constraints are satisfied. Deciding the existence of a solution for a CSP is NP-complete.

The dual encoding of a CSP $\mathcal{P}$ is a binary CSP, $\mathcal{P}^D$, where the variables are the relations of $\mathcal{P}$, and their domains are the tuples of those relations. A constraint exists between two variables in $\mathcal{P}^D$ if their corresponding relations' scopes intersect, and enforces equality over the shared variables. The graphical representation of $\mathcal{P}^D$ is the *dual graph*. In the dual graph, an edge between two vertices is redundant if there exists an alternate path between the two vertices such that the shared variables appear in every edge in the path [Janssen *et al.*, 1989; Dechter, 2003]. Redundant edges can be removed without changing the set of solutions. A *minimal dual graph*, a dual graph with no redundant edges, can be efficiently computed [Janssen *et al.*, 1989], but is not unique.

The *incidence graph* of a CSP is a bipartite graph where one set of vertices contains the variables of the CSP and the other set the constraints. An edge connects a variable and a constraint if and only if the variable appears in the scope of the constraint. The incidence graph is the same graph used in the hidden-variable encoding [Rossi *et al.*, 1990].

2.2 Consistency Properties and Algorithms

A CSP is globally consistent iff "any consistent instantiation of a subset of the variables can be extended to a consistent instantiation of all the variables without encountering any dead-ends" [Dechter, 2003]. Because guaranteeing a globally consistent CSP is in general exponential in time and space [Bessiere, 2006], we focus in practice on local consistency properties, which are in general tractable. Local consistency properties can be enforced on a CSP to filter values from the variables' domains or tuples from the constraints' relations to reduce the size of the search space. These properties can be enforced either before (i.e., pre-processing) or during search. Below, we informally review the consistency properties discussed in this paper.

The most common property is Arc Consistency (AC) for binary CSPs, or Generalized Arc Consistency (GAC) for non-binary CSPs [Mackworth, 1977]. A CSP is GAC iff, for every constraint c_i, and every variable $x \in scope(c_i)$, every value $v \in dom(x)$ is consistent with c_i (i.e., appears in some consistent tuple of R_i). Pairwise Consistency (PWC) is a relational consistency property that guarantees that every tuple consistent with a constraint c_i is consistent with every constraint in $neigh(c_i)$ [Gyssens, 1986]. Singleton Arc-Consistency (SAC) ensures that the CSP remains arc consistent after assigning a value to a variable [Debruyne and Bessière, 1997]. Neighborhood SAC (NSAC), a localized form of SAC, ensures that, for any assignment of a value to a variable x, the CSP induced by x and its neighborhood is arc consistent [Wallace, 2015]. Neighborhood Inverse Consistency (NIC) ensures that each value in the domain of a variable x can be extended to a solution in the subproblem induced by x and its neighborhood [Freuder and Elfe, 1996].

A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is Partition-One Arc-Consistent (POAC) iff $\mathcal{P}$ is SAC and for all $x_i \in \mathcal{X}$, for all $v_i \in dom(x_i)$, for all $x_j \in X$, there exists $v_j \in dom(x_j)$ such that $(x_i, v_i) \in AC(\mathcal{P} \cup \{x_j \leftarrow v_j\})$, where $AC(\mathcal{P} \cup \{x_j \leftarrow v_j\})$ is the CSP after assigning $x_j \leftarrow v_j$ and running AC [Bennaceur and Affane, 2001].

Using the terminology of Debruyne and Bessière [1997], we say that a consistency property p is *stronger* than p' if in any CSP where p holds p' also holds. Further, we say that p is *strictly stronger* than p' if p is stronger than p', and there exists at least one CSP in which p' holds but p does not. We say that p and p' are equivalent if p is stronger than p', and vice versa. Finally, we say that p and p' are incomparable when there exists at least one CSP in which p holds but p' does not, and vice versa. In practice, when a consistency property p is stronger than another p', enforcing p never yields less pruning than enforcing p' on the same problem.

Following this terminology, POAC is strictly stronger than SAC, which is strictly stronger than GAC. Further, NIC is strictly stronger than NSAC [Wallace, 2015].

POAC-1, the algorithm proposed by Balafrej *et al.* [2014] for enforcing POAC, operates by enforcing SAC while using counters on domain values to determine when more filtering can be obtained. They also propose APOAC, an adaptive version of POAC-1. In POAC-1, all the CSP variables are singleton tested, and the process is repeated over all the variables until a fixed point is reached. In APOAC, the first iteration is interrupted as soon as a given number of variables is processed. This number is learned and updated during search. Although not discussed by the authors, the consistency property enforced by the adaptive algorithm APOAC is strictly stronger than GAC, incomparable with SAC, and strictly weaker than POAC.

2.3 Minimum Cycle Basis

The composition of two cycles is the symmetric difference (exclusive-or) between the edges of the cycles. A *cycle basis* of a graph is a maximal set of cycles that are linearly independent (i.e., cycles in the basis cannot be obtained by taking the composition of other cycles in the basis) [Horton, 1987]. In a weighted graph the weight of a cycle in the graph is the sum of the weights of the edges in the cycle. A minimum cycle basis is a cycle basis where the sum of the weights of the cycles in the cycle basis is minimum. Informally, a minimum cycle basis is a minimum set of cycles that can generate all of the cycles of the graph. In the case of an unweighted graph, the weights of each edge is one, a minimum cycle basis has a minimum total length.[1] Algorithms for finding a minimum cycle basis are either exact or approximate, finding the minimum within some bound [Horton, 1987; Kavitha *et al.*, 2007; Mehlhorn and Michail, 2009; Amaldi *et al.*, 2010]. The complexity of the exact algorithm is $\mathcal{O}(e^2 n/\log(n))$ where n is the number of vertices and e the number of edges in the graph [Amaldi *et al.*, 2010]. That of the approximate algorithm is $\mathcal{O}(e^{\omega}\sqrt{n\log(n)})$ where ω is the best exponent of matrix multiplication ($\omega < 2.376$) [Kavitha *et al.*, 2007].

Figure 1 shows the incidence graph of a CSP, where the circles denote the variables and the squares the constraints. This graph has three cycles: (B, ABC, C, CD, D, BD), (C, CD, D, DF, F, EF, E, CE), and (B, ABC, C, CE, E, EF, F, DF, D, BD). The third cycle can be obtained from the first two by symmetric difference. Thus, the first two cycles constitute a minimal cycle basis for this graph. Incidentally, note that variable A does not appear in any cycle.

[1]Note that an MCB is not unique.

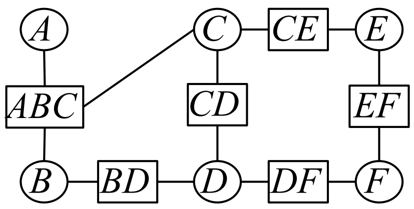

Figure 1: The incidence graph of a CSP

3 Localizing POAC

The algorithm POAC-1, which enforces POAC, runs a singleton test on *each* variable-value pair of the CSP [Balafrej *et al.*, 2014]. In each test, it enforces arc consistency on the *entire* CSP. Whenever the domain of *any* variable is updated, the entire process is repeated (i.e., POAC-1 runs the singleton test on all the variables again). We propose to reduce the cost of POAC-1 in two ways. First, at a singleton test on a given variable x, we restrict arc consistency to the variables in the cycles in which x appears. Second, whenever the domain of any variable, x or a variable that appears in a cycle of x, is updated as the result of this test, we repeat the singleton tests on all the variables in the cycles of the affected variable.

Incidentally, note that SAC and POAC are equivalent on cycles. Indeed, consider the network of the binary CSP shown in Figure 2. A singleton test on any of

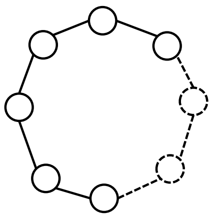

Figure 2: The constraint graph of the CSP is a cycle

the variables breaks the cycle into a chain and arc consistency guarantees global consistency [Freuder, 1982].

Proposition 1 *POAC is equivalent to SAC on a cycle.*

Sketch of proof: After breaking the cycle with a singleton test, arc consistency ensures global consistency. □

Below, we formalize the consistency property that results from our approach, then introduce the algorithms NPOACQ and $\cup_{cyc}$POACQ (Algorithms 1 and 5), which implements our idea. Then, we extend, in a trivial manner, our approach to relations. Finally, we discuss the use of POAC algorithms during search.

3.1 NPOAC: Localization to Neighborhoods

We define Neighborhood Partition-One Arc-Consistency (NPOAC) similarly to neighborhood SAC (NSAC) [Wallace, 2015]. Informally, neighborhood POAC localizes the singleton test to the neighborhood of the variable. Given a CSP $\mathcal{P}$ and $\mathcal{V}$ a subset of the variables of $\mathcal{P}$, we denote $\mathcal{P}|_{\mathcal{V}}$ the subproblem induced by $\mathcal{V}$ on $\mathcal{P}$. The constraints included in $\mathcal{P}|_{\mathcal{V}}$ are all those constraints whose scope contains a variable in $\mathcal{V}$.

Definition 1 *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is* Neighborhood Partition-One Arc-Consistent (NPOAC) *iff $\mathcal{P}$ is neighborhood SAC (NSAC), and for all $x_i \in \mathcal{X}$, for all $x_j \in neigh(x_i)$, for all $v_j \in dom(x_j)$, there exists $v_i \in dom(x_i)$ such that $v_j \in AC(\mathcal{P}|_{\{x_i\} \cup neigh(x_i)} \cup \{x_i \leftarrow v_i\})$.*

Theorem 2 *Neighborhood Inverse Consistency (NIC) is incomparable to Neighborhood POAC (NPOAC).*

Proof: Figure 3 shows a CSP that is NPOAC but variable v is not NIC. Figure 4 shows a CSP that is NIC but $X_4 \leftarrow 1$ is not NPOAC ($X_4 \leftarrow 1$ is removed in every singleton test for X_1). This example was first proposed to show that POAC is strictly stronger than SAC [Bennaceur and Affane, 2001]. □

3.2 $\cup_{cyc}$POAC: Localization to MCBs

For each singleton test for a given variable x_i, we propose to enforce arc consistency on the subproblem induced by the union of the variables of a minimum cycle basis (MCB) of x_i, where a MCB of x_i is computed on the incidence graph of the CSP.

First, we introduce some notations. Let MCB denote the set of cycles of a minimum cycle basis of the constraint network. For any given cycle $\phi \in MCB$, we denote $vars(\phi)$ the set of variables that appear in ϕ. For a given variable x_i, we denote $MCB(x_i) \subseteq MCB$ the set of cycles in which x_i appears. We define $vars(MCB(x_i))$ as follows:

$$vars(MCB(x_i)) = \{x_i\} \cup neigh(x_i) \cup_{\phi \in MCB(x_i)} vars(\phi).$$

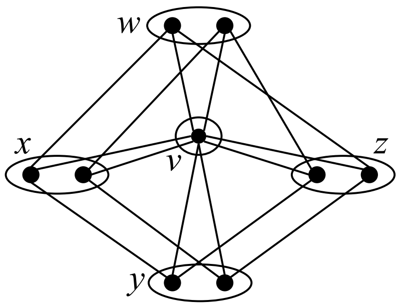

Figure 3: NPOAC but not NIC

R_{12}	
X_1	X_2
v_1	v_1
v_2	v_2
v_3	v_1
v_3	v_2

R_{13}	
X_1	X_3
v_1	v_1
v_1	v_3
v_2	v_2
v_2	v_3
v_3	v_1
v_3	v_2
v_3	v_3

R_{14}	
X_1	X_4
v_1	v_1
v_1	v_2
v_2	v_1
v_2	v_2
v_3	v_2

R_{23}	
X_2	X_3
v_1	v_2
v_1	v_3
v_2	v_1
v_2	v_3

R_{34}	
X_3	X_4
v_1	v_1
v_1	v_2
v_2	v_1
v_2	v_2
v_3	v_2

Figure 4: NIC but not NPOAC

This definition allows us to include variables that do not appear in a cycle (e.g., A does not appear in any cycle in Figure 1).

Now, we formulate the consistency property Union-Cycle Partition-One Arc-Consistency ($\cup_{cyc}$POAC). It is similar to POAC but restricts the propagation of arc consistency during a singleton test for a variable x_i to the subproblem induced on the CSP by the variables in $vars(MCB(x_i))$. Like POAC, the property must hold for all the variables of the CSP.

Definition 2 *Given a minimum-cycle basis MCB of a CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, $\mathcal{P}$ is* Union-Cycle Partition-One Arc-Consistent ($\cup_{cyc}$POAC) *iff for all $x_i \in \mathcal{X}$, the CSP $\mathcal{P}$ is AC for all $v_i \in dom(x_i)$ on $\mathcal{P}|_{vars(MCB(x_i))} \cup \{x_i \leftarrow v_i\}$, and for all $x_j \in vars(MCB(x_i))$, for all $v_j \in dom(x_j)$, there exists $v_i \in dom(x_i)$ such that $v_j \in AC(\mathcal{P}|_{vars(MCB(x_i))}) \cup \{x_i \leftarrow v_i\})$.*

It is easy to see that $\cup_{cyc}$POAC is strictly stronger than GAC, not comparable with SAC, and strictly weaker than POAC.

3.3 NPOACQ: A Variable-Based Algorithm

POAC-1, the original algorithm for POAC, uses a list of all the CSP variables, ordered by some heuristic such as decreasing values of dom/wdeg [Balafrej *et al.*, 2014]. After processing once every variable in the list, it repeats the process again whenever any domain is updated. Importantly, POAC-1 does not reconsider any variable for singleton testing before all the variables of the CSP have been processed. The size of the list does not change. To implement a similar behavior, our algorithm NPOACQ (Algorithm 1) uses three queues: Q stores the variables to be processed by singleton testing, Q_{seen} stores the variables that have been processed during the current iteration, and $Q_{toRevisit}$ stores the variables affected by change during the current iteration. Only when all the variables in Q have been processed (Q is empty), the variables in $Q_{toRevisit}$ are moved to Q to be processed.

Q is handled as a priority list using the same heuristic as POAC-1 (Line 4 of Algorithm 1). The popped variable is stored in Q_{seen} (Line 5) so that no variable is re-processed for singleton testing before Q is empty. `var`NPOACQ (Algorithm 3) is then called (Line 6 of Algorithm 1) to execute singleton tests for the popped variable. In Lines 9 and 19, `var`NPOACQ calls REQUEUE (Algorithm 2) on all the variables in the neighborhood of any variable whose domain was updated. REQUEUE adds those variables to $Q_{toRevisit}$ in case they were already singleton tested during the current iteration (Line 1), otherwise it adds them to Q (Line 2). When Q is empty, the variables in $Q_{toRevisit}$ are moved to Q, and Q_{seen} is cleared (Lines 7 and 8 of Algorithm 1).

Algorithm 1: NPOACQ($\mathcal{P}$)

```
  Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance
  Output: $true$ when $\mathcal{P}$ is $\cup_{cyc}$POAC, otherwise $false$
1 $Q \leftarrow \mathcal{V}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
2 $consistent \leftarrow$ ENFORCEAC($\mathcal{P}, \emptyset$)
3 while $consistent$ and $Q \neq \emptyset$ do
4     $x_i \leftarrow$ POP($Q$)
5     $Q_{seen} \leftarrow Q_{seen} \cup \{x_i\}$
6     if not varNPOACQ($x_i, \mathcal{P}$) then return $false$
7     if $Q = \emptyset$ and $Q_{toRevisit} \neq \emptyset$ then
8         $Q \leftarrow Q_{toRevisit}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
9 return $true$
```

Algorithm 2: REQUEUE(x_i)

```
  Input: $x_i$: a variable to requeue
  Output: Adds $x_i$ to either $Q$ or $Q_{toRevisit}$
1 if $x_i \in Q_{seen}$ then $Q_{toRevisit} \leftarrow Q_{toRevisit} \cup \{x_i\}$
2 else $Q \leftarrow Q \cup \{x_i\}$,
```

`var`NPOACQ (Algorithm 3) runs singleton tests on a given CSP variable by calling TESTAC (Algorithm 4) which enforces arc consistency on the subproblem induced on the CSP by the variables in $neigh(x_i)$ (Line 4). As in POAC-1, whenever a value is removed from a variable's domain, `var`NPOACQ enforces AC on the CSP (Lines 6 and 20).

Algorithm 3: `var`NPOACQ($x_i, \mathcal{P}$)

```
Input: $x_i$: Variable to instantiate; $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance; $Q$: The
        propagation queue
Output: $true$ if consistent, else $false$
1  $\forall x_j \in \mathcal{V}, v_j \in dom(x_j), counter(x_j, v_j) \leftarrow 0$
2  $size \leftarrow |dom(x_i)|$
3  foreach $v_i \in dom(x_i)$ do
4      if not TESTAC($\{x_i\} \cup neigh(x_i), \mathcal{D}, cons(x_i) \cup \{x_i \leftarrow v_i\}, counter(\cdot,\cdot)$)
        then
5          $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
6          if not ENFORCEAC($\mathcal{P}, L$) then return $false$
7  if $dom(x_i) = \emptyset$ then return $false$
8  if $|dom(x_i)| \neq size$ then
9      foreach $x_k \in neigh(x_i)$ do REQUEUE($x_k$)
10 $change \leftarrow false$
11 foreach $x_j \in neigh(x_i)$ do
12     $size \leftarrow |dom(x_j)|$
13     foreach $v_j \in dom(x_j)$ do
14         if $counter(x_j, v_j) = |dom(x_i)|$ then
15             $dom(x_j) \leftarrow dom(x_j) \setminus \{v_j\}$, $change \leftarrow true$
16         $counter(x_j, v_j) \leftarrow 0$
17     if $dom(x_j) = \emptyset$ then return $false$
18     if $|dom(x_j)| \neq size$ then
19         foreach $x_k \in neigh(x_j) \setminus \{x_i\}$ do REQUEUE($x_k$)
20 if $change$ and not ENFORCEAC($\mathcal{P}, \emptyset$) then return $false$
21 return $true$
```

Like POAC-1, we use the data structure $counter(\cdot,\cdot)$. $counter(x_j, v_j)$ records how many times value v_j of variable x_j was pruned during the singleton tests for another variable x_i. If, after running all the singleton tests for x_i, $counter(x_j, v_j) = |dom(x_i)|$, then we know that (x_j, v_j) is necessarily inconsistent and can be safely removed. TESTAC (Algorithm 4) implements the singleton test for $x_i \leftarrow v_i$ and updates $counter(\cdot,\cdot)$. ENFORCEAC($\mathcal{P}, L$) allows running *any* arc consistency algorithm. It stores in L the list of variable-value pairs that were removed as a result of enforcing AC. TESTAC (Algorithm 4) updates the counters only when the problem is arc consistent (Line 6).

```
Algorithm 4: TESTAC($\mathcal{P}, counter(\cdot,\cdot)$)
   Input: $\mathcal{P}$: A CSP instance; $counter(\cdot,\cdot)$: the counter data structure
   Output: $true$ if consistent, else $false$
 1 $L \leftarrow \emptyset$
 2 $consistent \leftarrow$ ENFORCEAC($\mathcal{P}$, $L$)
 3 foreach $(x_j, v_j) \in L$ do
 4     $dom(x_j) \leftarrow dom(x_j) \cup \{v_j\}$
 5     if $consistent$ then
 6         $counter(x_j, v_j) \leftarrow counter(x_j, v_j) + 1$
 7 return $consistent$
```

3.4 $\cup_{cyc}$POACQ: A Variable-Based Algorithm

$\cup_{cyc}$POACQ (Algorithm 5 is similar to NPOACQ (Algorithm 1). The major difference is in Line 6, where `var`$\cup_{cyc}$POACQ (Algorithm 6) is called to execute singleton tests for the popped variable.

`var`$\cup_{cyc}$POACQ (Algorithm 6) is similar to `var`NPOACQ (Algorithm 3), which runs singleton tests on a given CSP. The major difference is in Line 4, where TESTAC is induced on the MCB of a variable, rather than its neighborhood. `var`$\cup_{cyc}$POACQ does not restrict how MCBs are generated (i.e., using exact or approximate algorithms) or the graphs (i.e., incidence or dual) on which they are computed.

3.5 Extension to Relations

First, we extend the definition of POAC to relations.

Definition 3 *A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is* Relational Partition-One Arc-Consistent (rPOAC) *iff the CSP is singleton PWC, and for all $c_i \in \mathcal{C}$, for all $\tau_i \in R_i$, for all $c_j \in \mathcal{C}$, there exists $\tau_j \in R_j$ such that $(c_i, \tau_i) \in PWC(\mathcal{P} \cup \{R_i \leftarrow \tau_i\})$.*

Algorithm 5: $\cup_{cyc}$POACQ($\mathcal{P}, MCB$)

```
Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance; $MCB$: a minimum cycle basis of $\mathcal{P}$
Output: $true$ when $\mathcal{P}$ is $\cup_{cyc}$POAC, otherwise $false$
1  $Q \leftarrow \mathcal{V}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
2  $consistent \leftarrow$ ENFORCEAC($\mathcal{P}, \emptyset$)
3  while $consistent$ and $Q \neq \emptyset$ do
4      $x_i \leftarrow$ POP($Q$)
5      $Q_{seen} \leftarrow Q_{seen} \cup \{x_i\}$
6      if not var$\cup_{cyc}$POACQ($x_i, \mathcal{P}, MCB$) then
7          return $false$
8      if $Q = \emptyset$ and $Q_{toRevisit} \neq \emptyset$ then
9          $Q \leftarrow Q_{toRevisit}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
10 return $true$
```

The algorithm to enforce rPOAC is a trivial adaptation of the variable-based POAC algorithm: it operates on relations' tuples instead of variables' values. Further, instead of AC, we enforce pair-wise consistency (PWC). We denote rN-POACQ and $\cup_{cyc}$rPOACQ the adaptation of NPOACQ and $\cup_{cyc}$POACQ, respectively, to relations.

3.6 Observations and Comments

Below, we make some observations useful in practice.

Singleton domains. Because the algorithms for enforcing POAC-like properties (e.g., POAC-1 and $\cup_{cyc}$POACQ) enforce GAC whenever singleton testing a variable yields a domain update, variables with a singleton domain never need to be singleton tested. We do not include this test in our pseudocode to avoid reducing readability.

Proposition 3 *On a CSP that is GAC, singleton testing a variable x with $|dom(x)| = 1$ yields no filtering.*

Proof: After assigning x to the unique value in its domain, the CSP remains GAC. □

Algorithm 6: $\texttt{var}\cup_{cyc}$POACQ($x_i, \mathcal{P}, MCB$)

```
Input: $x_i$: Variable to instantiate; $\mathcal{P}$: A CSP instance; $MCB$: a minimum cycle
       basis of $\mathcal{P}$
Output: $true$ if consistent, else $false$
1  $\forall x_j \in \mathcal{V}, v_j \in dom(x_j), counter(x_j, v_j) \leftarrow 0$
2  $size \leftarrow |dom(x_i)|$
3  foreach $v_i \in dom(x_i)$ do
4      if not TESTAC($\mathcal{P}|_{vars(MCB(x_i))} \cup \{x_i \leftarrow v_i\}, counter(\cdot,\cdot)$) then
5          $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
6          if not ENFORCEAC($\mathcal{P}, \emptyset$) then return $false$
7  if $dom(x_i) = \emptyset$ then return $false$
8  if $|dom(x_i)| \neq size$ then
9      foreach $x_k \in vars(MCB(x_i)) \setminus \{x_i\}$ do REQUEUE($x_k$)
10 $change \leftarrow false$
11 foreach $x_j \in vars(MCB(x_i)) \setminus \{x_i\}$ do
12     $size \leftarrow |dom(x_j)|$
13     foreach $v_j \in dom(x_j)$ do
14         if $counter(x_j, v_j) = |dom(x_i)|$ then
15             $dom(x_j) \leftarrow dom(x_j) \setminus \{v_j\}$, $change \leftarrow true$
16         $counter(x_j, v_j) \leftarrow 0$
17     if $dom(x_j) = \emptyset$ then return $false$
18     if $|dom(x_j)| \neq size$ then
19         foreach $x_k \in vars(MCB(x_j)) \setminus \{x_j\}$ do REQUEUE($x_k$)
20 if $change$ and not ENFORCEAC($\mathcal{P}, \emptyset$) then return $false$
21 return $true$
```

Domino effect. This observation allows us, during backtrack search using a POAC-like algorithm for real-full lookahead, to instantiate all variables with singleton domains (i.e., domino effect) without re-enforcing consistency because no further filtering can be obtained, thus saving on effort. Note that the same behavior is implicitly guaranteed for consistency algorithms using supports.

Q initialization. After an assignment $x \leftarrow v$ during search, Q is initialized to $vars(MCB(x)) \setminus \{x\}$.

Large variable domains. On small variable domains singleton testing is *quicker* and empirically yields *more filtering* than on larger variables' domains. The observation explains why using dom/wdeg to order the variables for singleton testing yields good performance in practice. It also explains the good performance of the adaptive algorithm APOAC, which avoids singleton testing variables with large domains, a costly process that rarely yields any filtering.

4 Experimental Evaluation

The goal of the section is to assess the effectiveness of localizing POAC to neighborhoods and cycles when used for real-full lookahead during search. To that end, we evaluate finding a single solution to a CSP using backtrack search, real-full lookahead, and the dom/wdeg variable ordering heuristic [Boussemart *et al.*, 2004]. We also use dom/wdeg to select the variable for singleton testing in the POAC-based algorithms.

We first compared NPOACQ (Algorithm 1) and $\cup_{cyc}$POACQ (Algorithm 5). $\cup_{cyc}$POACQ outperformed NPOACQ on the majority of tested problems. Thus we consider only $\cup_{cyc}$POACQ in our evaluation.

We study the performance of the following variable-based algorithms: **GAC** (GAC2001 [Bessière *et al.*, 2005]), **POAC** (POAC-1 [Balafrej *et al.*, 2014]), **APOAC** (adaptive POAC [Balafrej *et al.*, 2014]), $\cup_{cyc}$**POACQ** (Section 3.2), and **A**$\cup_{cyc}$**POACQ** (an adaptive version of our new algorithm). The MCB is computed on the incidence graph. For the relational versions, we study the performance of the following algorithms: **PW-AC2** (an improved version of the algorithm for pairwise consistency of Samaras and Stergiou [2005]), **rPOAC**, **ArPOACQ**, $\cup_{cyc}$**rPOACQ**, and **A**$\cup_{cyc}$**rPOACQ**. The MCB is computed on a minimal dual graph.

For pre-processing, we run POAC-1 until quiescence for the non-adaptive versions of the algorithms and POAC-1 once through all the variables for the adaptive versions. This strategy was advocated as the most effective by Balafrej *et al.* [2014]. The singleton testing in the adaptive algorithms is interrupted after a given number of variables has been processed. This cutoff values is learned during search. We use the best adaptive version reported by Balafrej *et al.* [2014], where the maximum number of singleton calls, $maxK$, is initialized to the number of variables in the problem. The algorithm spends 1/10 of its time learning[2] a $maxK$ threshold and 9/10 of its time exploiting the learned $maxK$.

[2]Using the terminology of Balafrej *et al.* [2014], $maxK = n$, last drop with $\beta = 0.05$, and 70%-PER.

We conducted the experiments on the following benchmark problems from Lecoutre's webpage:[3] TSP-25, ukVg, QCP-15, cril, QWH-20, k-insertions, mug, TSP-20, renault, myciel, lexVg, varDimacs, and rand-2-40-19. Those benchmark include all those reported by Balafrej *et al.* [2014]. We set a time limit of four hours per instance with 8GB of memory.

Table 1 shows the average CPU time in seconds to generate an MCB for the evaluated benchmarks.[4] This time is insignificant except for the following benchmarks: QCP-15, cril, QWH-20, k-insertions, myciel, and varDimacs. We believe that the algorithm used [Amaldi *et al.*, 2010], written for weighted graphs, can be improved for unweighted graphs. Also, one could look for alternative techniques for generating cycles. This issue deserves further investigation. The time to generate the cycles is included in the results of Tables 2 and 3.

Table 1: Average CPU time (seconds) to generate MCB

	TSP-25	ukVg	QCP-15	cril	QWH-20	k-insertions	mug	TSP-20	renault	myciel	lexVg	varDimacs	rand-2-40-19
incidence	0.07	0.09	6.56	6.58	36.85	21.42	0.02	0.03	0.03	1.39	0.06	6.76	0.08
min. dual	0.11	0.01	17.67	62.34	116.79	82.74	0.01	0.04	0.03	20.70	0.00	3.64	0.29

In Tables 2 and 3, we report, for each benchmark, the total number of instances in the benchmark (total) and the number of instances completed by at least one algorithm (solv-by-1). For each algorithm, we report the number of instances solved (# solv) and the sum of the CPU time in seconds (ΣCPU (s)) computed over the instances of solv-by-1. When an algorithm does not terminate within four hours, we add 14,400 seconds to the CPU time, and indicate with a $>$ sign that the time reported is a lower bound. A value is bold face in the table if it is the best value in the row.

4.1 Variable-Based Algorithms

This experiment shows that, when a POAC-like algorithm exploits cycles, its performance is almost always improved. Comparing columns APOAC and A$\cup_{cyc}$POACQ,

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

[4]We extended the memory allocation to 16GB for the generation of the MCBs of the k-insertion and QWH-20 benchmarks.

we notice that A$\cup_{cyc}$POACQ is faster than, and solves at least as many instances as, APOAC except on k-insertions and mug. Concerning k-insertions, Table 1 shows that the amount of time to generate the MCB is not insignificant. This cost partially explains the CPU increase over APOAC. As for mug, the adaptive algorithms were already shown in the original paper not to win over the non adaptive versions [Balafrej *et al.*, 2014]. Further, our new algorithm $\cup_{cyc}$POACQ, beats POAC on mug and shows the best overall performance on this benchmark. Moreover, note that on the first three benchmarks of Table 1 (i.e., TSP-25, ukVg, QCP-15), APOAC is actually *slower* than GAC. However, our algorithm A$\cup_{cyc}$POACQ outperforms both GAC and APOAC.

As was highlighted by Balafrej *et al.* [2014], there are benchmarks where GAC remains the winner (e.g., TSP-20, renault, myciel, lexVg, varDimacs, and rand-2-40-19). However, A$\cup_{cyc}$POACQ reduces the gap between GAC and APOAC.

Table 2: Lookahead with POAC-based algorithms (MCB on the incidence graph)

Benchmark	**# Instances**		**GAC**	**POAC**	**$\cup_{cyc}$POACQ**	**APOAC**	**A$\cup_{cyc}$POACQ**
TSP-25	total: 15	# solv	**15**	14	**15**	**15**	**15**
	solv-by-1: 15	ΣCPU (s)	4,303.12	>41,382.27	32,654.67	6,152.91	**2,418.41**
ukVg	total: 65	# solv	40	34	25	38	**41**
	solv-by-1: 41	ΣCPU (s)	>52,776.77	>134,534.24	>253,718.99	>72,950.60	**36,137.13**
QCP-15	total: 15	# solv	**15**	**15**	**15**	**15**	**15**
	solv-by-1: 15	ΣCPU (s)	1,172.59	4,704.29	2,513.27	1,237.20	**948.35**
cril	total: 8	# solv	6	7	7	**8**	**8**
	solv-by-1: 8	ΣCPU (s)	>30,458.10	>16,282.45	>16,651.04	2,321.96	**1,831.60**
QWH-20	total: 10	# solv	**10**	**10**	**10**	**10**	**10**
	solv-by-1: 10	ΣCPU (s)	2,256.61	6,154.43	3,007.98	2,236.32	**2,061.63**
k-insertions	total: 32	# solv	17	17	**18**	**18**	**18**
	solv-by-1: 18	ΣCPU (s)	>17,034.30	>21,639.31	11,814.83	**6,129.92**	8,940.59
mug	total: 8	# solv	6	6	**8**	6	6
	solv-by-1: 8	ΣCPU (s)	>54,724.38	>29,385.02	**13,655.87**	>34,207.98	>41,583.97
TSP-20	total: 15	# solv	**15**	**15**	**15**	**15**	**15**
	solv-by-1: 15	ΣCPU (s)	**302.21**	2,750.90	3,096.07	593.04	384.13
renault	total: 50	# solv	**50**	**50**	**50**	**50**	**50**
	solv-by-1: 50	ΣCPU (s)	**55.87**	277.74	176.28	196.04	155.88
myciel	total: 16	# solv	**13**	12	12	**13**	**13**
	solv-by-1: 13	ΣCPU (s)	**1,711.93**	>21,564.06	>26,196.15	3,118.86	2,555.54
lexVg	total: 63	# solv	**63**	**63**	60	**63**	**63**
	solv-by-1: 63	ΣCPU (s)	**3,413.70**	20,652.58	>109,938.00	6,242.65	4,703.04
varDimacs	total: 9	# solv	**9**	8	8	**9**	**9**
	solv-by-1: 9	ΣCPU (s)	**4,116.43**	>15,208.70	>15,614.45	5,203.18	4,406.16
rand-2-40-19	total: 50	# solv	**50**	36	42	**50**	**50**
	solv-by-1: 50	ΣCPU (s)	**20,591.17**	>339,527.03	>289,945.67	65,622.27	43,710.03

4.2 Relation-Based Algorithms

Comparing Tables 2 and 3, we notice that the variable-based algorithms outperform the relational versions except for ukVg and lexVg (both crossword puzzles), where PW-AC2 is the fastest algorithm.

Table 3: Search results using rPOAC-based algorithms for lookahead (MCB computed on the minimal dual graph)

Benchmark	# Instances		PW-AC2	rPOAC	$\cup_{cyc}$rPOACQ	ArPOAC	A$\cup_{cyc}$rPOACQ
TSP-25	total: 15	# solv	**15**	5	5	14	**15**
	solv-by-1: 15	ΣCPU (s)	**12,223.66**	>151,095.28	>152,503.48	>32,178.91	13,474.89
ukVg	total: 65	# solv	**41**	24	25	24	25
	solv-by-1: 41	ΣCPU (s)	**27,756.04**	>270,995.87	>268,324.95	>268,850.65	>266,815.65
QCP-15	total: 15	# solv	14	**15**	**15**	**15**	14
	solv-by-1: 15	ΣCPU (s)	>18,111.36	17,157.06	33,854.40	**8,159.52**	>19,405.52
cril	total: 8	# solv	4	**6**	6	5	6
	solv-by-1: 7	ΣCPU (s)	>47,413.74	>**20,737.10**	>28,066.46	>34,663.42	>22,413.73
QWH-20	total: 10	# solv	**10**	9	8	**10**	**10**
	solv-by-1: 10	ΣCPU (s)	**9,629.87**	>48,522.86	>60,889.81	25,976.61	22,650.42
k-insertions	total: 32	# solv	**17**	14	16	**17**	15
	solv-by-1: 18	ΣCPU (s)	>**21,043.39**	>67,578.37	>46,384.72	>24,622.72	>45,181.08
mug	total: 8	# solv	4	**7**	5	6	4
	solv-by-1: 7	ΣCPU (s)	>43,200.11	**11,297.27**	>29,507.67	>25,399.39	>43,200.51
TSP-20	total: 15	# solv	**15**	14	13	**15**	**15**
	solv-by-1: 15	ΣCPU (s)	**531.92**	>33,332.30	>60,507.05	2,834.57	1,950.34
renault	total: 50	# solv	**50**	41	42	**50**	**50**
	solv-by-1: 50	ΣCPU (s)	**252.05**	>168,307.39	>165,457.29	57,844.36	61,277.01
myciel	total: 16	# solv	**13**	9	10	**13**	**13**
	solv-by-1: 13	ΣCPU (s)	**5,307.93**	>62,439.79	>50,673.50	8,132.70	6,380.32
lexVg	total: 63	# solv	**63**	58	55	59	58
	solv-by-1: 63	ΣCPU (s)	**3,341.66**	>128,012.64	>156,491.63	>107,738.42	>128,771.50
varDimacs	total: 9	# solv	**9**	7	7	**9**	**9**
	solv-by-1: 9	ΣCPU (s)	**7,371.85**	>30,308.20	>30,467.75	10,848.36	9,461.34
rand-2-40-19	total: 50	# solv	49	0	0	12	**50**
	solv-by-1: 50	ΣCPU (s)	>132,509.05	>720,000.00	>720,000.00	>574,934.46	**22,574.42**

Examining Table 3, we notice that PW-AC2 is in general the fastest and can solve the most benchmarks (except for QCP-15, cril, mug, and rand-2-40-19). Further, we see that the adaptive mechanism is useful: both ArPOAC and A$\cup_{cyc}$rPOACQ are faster than rPOAC and $\cup_{cyc}$rPOACQ (except on cril and mug). We strongly believe that the adaptive mechanism could be further improved with better tuning of the parameters, which is beyond the topic of this paper. Regardless, A$\cup_{cyc}$rPOACQ still outperforms ArPOAC on the majority of the benchmarks (except on QCP-15, k-insertions, mug, renault, and lexVg). Most notably, A$\cup_{cyc}$rPOACQ outperforms all relational algorithms, including PW-AC2, on rand-2-40-19, which is encouraging. While it is too early to rule out the usefulness of the relational versions of POAC, the effectiveness of propagation over cycles is noteworthy even in this context.

5 Future Work & Conclusions

In this paper, we advocate the use of cycles to improve the performance of algorithms for enforcing POAC and provide empirical evidence of the benefit of our approach. Future work should improve the performance of the algorithms for computing MCBs, and extend our approach to other consistency algorithms.

Acknowledgments: This research is supported by NSF Grant No. RI-111795 and RI-1619344. Experiments were conducted on the equipment of the Holland Computing Center at the University of Nebraska-Lincoln.

References

[Amaldi *et al.*, 2010] Edoardo Amaldi, Claudio Iuliano, and Romeo Rizzi. Efficient Deterministic Algorithms for Finding a Minimum Cycle Basis in Undirected Graphs. In *Integer Programming and Combinatorial Optimization (IPCO 2010)*, volume 6080 of *LNCS*, pages 397–410, 2010.

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

[Bennaceur and Affane, 2001] Hachemi Bennaceur and Mohamed-Salah Affane. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Principles and Practice of Constraint Programming (CP 2001)*, volume 2239 of *LNCS*, pages 560–564, 2001.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Bessiere, 2006] Christian Bessiere. *Handbook of Constraint Programming*, chapter Constraint Propagation, pages 29–83. Elsevier, 2006.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proc. ECAI 2004*, pages 146–150, 2004.

[Debruyne and Bessière, 1997] Romuald Debruyne and Christian Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15 th International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Freuder and Elfe, 1996] Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI-96*, pages 202–208, Portland, Oregon, 1996.

[Freuder, 1982] Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.

[Gyssens, 1986] M. Gyssens. On the Complexity of Join Dependencies. *ACM Trans. Database Systems*, 11(1):81–108, 1986.

[Horton, 1987] Joseph D. Horton. A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[Janssen *et al.*, 1989] P. Janssen, Philippe Jégou, B. Nougier, and M.C. Vilarem. A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In *IEEE Workshop on Tools for AI*, pages 420–427, 1989.

[Kavitha *et al.*, 2007] Telikepalli Kavitha, Kurt Mehlhorn, and Dimitrios Michail. New Approximation Algorithms for Minimum Cycle Bases of Graphs. In *Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *LNCS*, pages 512–523, 2007.

[Mackworth, 1977] Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mehlhorn and Michail, 2009] Kurt Mehlhorn and Dimitrios Michail. Minimum Cycle Bases: Faster and Simpler. *ACM Trans. Algorithms*, 6(1):1–13, December 2009.

[Rossi *et al.*, 1990] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 550–556, 1990.

[Samaras and Stergiou, 2005] Nikos Samaras and Kostas Stergiou. Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. *Journal of Artificial Intelligence Research*, 24:641–684, 2005.

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