Arc Consistency during Search

Chavalit Likitvivatanavong
School of Computing
National University of Singapore

Yuanlin Zhang
Scott Shannon
Dept. of Computer Science
Texas Tech University, USA

James Bowen
Eugene C. Freuder
Cork Constraint Computation Centre
University College Cork, Ireland

Abstract

Enforcing arc consistency (AC) during search has proven to be a very effective method in solving Constraint Satisfaction Problems and it has been widely-used in many Constraint Programming systems. Although much effort has been made to design efficient standalone AC algorithms, there is no systematic study on *how* to *efficiently* enforce AC *during search*, as far as we know. The significance of the latter is clear given the fact that AC will be enforced millions of times in solving hard problems. In this paper, we propose a framework for enforcing AC during search (ACS) and complexity measurements of ACS algorithms. Based on this framework, several ACS algorithms are designed to take advantage of the residual data left in the data structures by the previous invocation(s) of ACS. The algorithms vary in the worst-case time and space complexity and other complexity measurements. Empirical study shows that some of the new ACS algorithms perform better than the conventional implementation of AC algorithms in a search procedure.

1 Introduction and background

Enforcing arc consistency (AC) on constraint satisfaction problems (CSP) during search has been proven very successful in the last decade [Sabin and Freuder, 1994; Mackworth, 1977] As AC can be enforced millions of times in solving hard instances, the need for efficient AC algorithms is obvious. Given the numerous attempts to optimize standalone AC algorithms, further improvement on their performance becomes very challenging. In this paper, in order to improve the overall efficiency of a search procedure employing arc consistency, we focus on *how to efficiently enforce AC during search (ACS)*, rather than on standalone AC algorithms.

In this paper, we abstract ACS into a separate module that maintains AC on a changing CSP problem P with four methods. Several complexity measurements are then proposed to evaluate the theoretical efficiency of ACS algorithms in term of these methods. A key method is ACS.try($x = a$), where $x = a$ is an assignment. It checks whether $P \cup \{x = a\}$ can be made arc consistent. Whenever a search procedure makes an assignment, it will call ACS.try() with that assignment as the argument and make further decision based on the return value of try(). With the explicit abstraction of ACS, we notice that after one invocation of an ACS method, say ACS.try(), there are *residual data* left in the structures of ACS. We will explore how to make use of these residual data to design new ACS algorithms with the new measurements in mind. Empirical study is also carried out to benchmark the new ACS algorithms and those designed using conventional techniques. One of the new ACS algorithms is very simple but shows a clear performance advantage (clock time) over the rest. Necessary background is reviewed below.

A *binary constraint satisfaction problem (CSP)* is a triple (V, D, C) where V is a finite set of variables, $D = \{D_x \mid x \in V$ and D_x is the finite domain of $x\}$, and C is a finite set of binary constraints over the variables of V. As usual, we assume there is at most one constraint on a pair of variables. We use n, e, and d to denote the number of variables, the number of constraints, and the maximum domain size of a CSP problem.

Given a constraint c_{xy}, a value $b \in D_y$ is a *support* of $a \in D_x$ if $(a, b) \in c_{xy}$, and a *constraint check* involves determining whether $(u, v) \in c_{xy}$ for some $u \in D_x$ and $v \in D_y$. A constraint c_{xy} is *arc consistent* if each value of D_x has a support in D_y and every value of D_y has a support in D_x. A CSP problem is *arc consistent (AC)* if all its constraints are arc consistent. To *enforce arc consistency* on a CSP problem is to remove from the domains the values that have no support. A CSP is *arc inconsistent* if a domain becomes empty when AC is enforced on the problem.

We use D_x^0 to denote the *initial domain* of x before the search starts while D_x the *current domain* at a moment during AC or search. A value u is *present in* (or *absent from*, respectively) D_x if $u \in D_x$ ($u \notin D_x$ respectively). For each domain $D_x \in D$, we introduce two dummy values head and tail. We assume there is a total ordering on $D_x \cup \{\mathsf{head}, \mathsf{tail}\}$ where head is the *first*, i.e., smallest, value, and tail the *last* (largest) value. For any a from D_x^0, $\mathsf{succ}(a, D_x)$ ($\mathsf{pred}(a, D_x)$ respectively) is the first (last respectively) value of $D_x \cup \{\mathsf{head}, \mathsf{tail}\}$ that is greater (smaller respectively) than a.

2 Enforcing arc consistency during search

We take a CSP solver as an iterative interaction between a search procedure and an ACS algorithm. The ACS algorithm

can be abstracted into one data component and four methods: a CSP problem P, init(P_1), try($x = a$), backjump($x = a$), and addInfer($x \neq a$), where P_1 is another CSP problem, $x \in P.V$, and $a \in P.D_x$. Throughout the paper, $P.V$, $P.D_x (x \in P.V)$, and $P.C$ denote the set of variables, the domain of x, and the set of constraints of P.

ACS.P is accessible (read only) to a caller. When the context is clear, we will simply use P, instead of ACS.P.

ACS.init(P_1) sets P to be P_1 and creates and initializes the internal data structures of ACS. It returns false if P is arc inconsistent, and true otherwise. ACS.try($x = a$) enforces arc consistency on $P_1 = P \cup \{x = a\}$. If the new problem P_1 is arc consistent, it sets P to be P_1 and returns true. Otherwise, $x = a$ is discarded, the problem P remains unchanged, and try() returns false. In general, the method can accept any type of constraints, e.g., $x \geq a$. ACS.addInfer($x \neq a$) enforces arc consistency on $P_1 = P \cup \{x \neq a\}$. If the new problem P_1 is arc consistent, it sets P to be P_1 and returns true. Otherwise, addInfer() returns false. When MAC infers that x can not take value a, it calls ACS.addInfer($x \neq a$). In general, any constraint can be added as long as it is inferred from the current assignments by the search procedure. ACS.backjump($x = a$) discards from P all constraints added, by ACS.try() or ACS.addInfer(), to P since (including) the addition of $x = a$. The consequences of those constraints caused by arc consistency processing are also retracted. This method does not return a value. We ignore the prefix ACS of a method if it is clear from the context.

A search procedure usually does not invoke the ACS methods in an arbitrary order. The following concept characterizes a rather typical way for a search procedure to use ACS methods. Given a problem P, *a canonical invocation sequence (CIS)* of ACS methods is a sequence of methods m_1, m_2, ..., m_k satisfying the following properties: 1) m_1 is init(P_1) and for any i ($2 \leq i \leq k$), $m_i \in$ {try(), addInfer(), backjump()}; 2) m_1 returns true if $k \geq 2$; 3) for any try($x = a$) and addInfer($x \neq a$) $\in \{m_2, \ldots, m_k\}$, $x \in$ ACS.$P.V$ and $a \in D_x$ at the moment of invocation; 4) for any m_i=backjump($y = a$) where $2 \leq i \leq k$, m_{i-1} must be an invocation of try() or addInfer() that returns false, and there exists m_j such that $2 \leq j < i - 1$ and m_j=try($y = a$) and there is no backjump($y = a$) between m_j and m_i; 5) for any m_i=addInfer() where $2 \leq i \leq k$, if it returns false, m_{i+1} must be backjump(). Note that an arbitrary canonical invocation sequence *might not* be a sequence generated by any meaningful search procedure.

Example Algorithm 1 (line 1–15) illustrates how MAC [Sabin and Freuder, 1994] can be designed using ACS.

2.1 Template implementation of ACS methods

To facilitate the presentation of our ACS algorithms, we list a template implementation for each ACS method in Algorithm 1. Since try() could change the internal data structures and the domains of the problem P, it simply backups the current state of data structures with timestamp(x, a) (line 18) before it enforces arc consistency (line 19–21). An alternative is to "backup the changes" which is not discussed here because it does not affect any complexity measures of ACS algorithms (except possibly the clock time). ACS-X.propagate() follows

Algorithm 1: MAC and template ACS methods

```
                    ——————————MAC algorithm——————————
MAC(P)
 1  if (not ACS.init(P)) then return no solution
 2  create an empty stack s; // no assignments is made yet
 3  freevariables ← P.V
 4  while (freevariables ≠ ∅) do
 5      Select a variable x_i from freevariables and a value a for x_i
 6      if (ACS.try(x_i = a)) then
 7          s.push((x_i, a))
 8          freevariables ← freevariables −{x_i}
 9      else
10          while (not ACS.addInfer(x_i ≠ a)) do
11              if (s is not empty) then (x_i, a) ← s.pop()
12              else return no solution
13              ACS.backjump(x_i = a)
14              freevariables ← freevariables ∪{x_i};
15  return the assignments
                    ——————————Template ACS methods——————————
ACS-X.init(P_1)
16  P ← P_1, initialize the internal data structures of ACS-X
17  return AC(P) // AC() can be any standalone AC algorithm
ACS-X.try(x = a)
18  backup(timestamp (x, a))
19  delete all values except a from D_x
20  Q ← {(y, x) | c_yx ∈ P.C }
21  if (propagate(Q)) then return true
22  else ACS-X.restore(timestamp (x, a)), return false
ACS-X.addInfer(x ≠ a)
23  delete a from D_x, Q ← {(y, x) | c_yx ∈ P.C}, return propagate(Q)
ACS-X.backjump(x = a)
24  restore(timestamp (x, a))
ACS-X.backup(timestamp (x, a))
25  backup the internal data structures of ACS-X, following timestamp (x, a)
26  backup the current domains, following timestamp (x, a)
ACS-X.restore(timestamp (x, a))
27  restore the internal data structures of ACS-X, following timestamp (x, a)
28  restore the domains of P, following timestamp (x, a)
ACS-X.propagate(Q)
29  while Q ≠ ∅ do
30      select and delete an arc (x, y) from Q
31      if revise(x, y) then
32          if D_x = ∅ then return false
33          Q ← Q ∪ {(w, x) | C_ux ∈ P.C, w ≠ y}
34  return true
ACS-X.revise(x, y)
35  delete ← false
36  foreach a ∈ D_X do
37      if not hasSupport(x, a, y) then
38          delete ← true, delete a from D_X
39  return delete
ACS-X.hasSupport(x, a, y)
40  if (∃b ∈ D_y such that (a, b) ∈ c_xy) then return true
41  else return false
```

that of AC-3 [Mackworth, 1977]. ACS-X.addInfer($x \neq a$) (line 23) does not call backup() because $x \neq a$ is an inference from the current assignments and thus no new backup is necessary.

2.2 Complexity of ACS algorithms

We present several types of the time and space complexities for ACS algorithms. The *node-forward time complexity* of an ACS algorithm is the worst-case time complexity of ACS.try($x = a$) where $x \in P.V$ and $a \in D_x$. An *incremental sequence* is a consecutive invocations of ACS.try() where each invocation returns true and no two invocations involve the the same variable (in the argument). The *path-forward time complexity* of an ACS is the worst-case time complexity of any incremental sequence with any $k \leq n$ (the size of $P.V$) invocations. *Node-forward space complexity* of an ACS algorithm is the worst case space complexity of the internal data structures (excluding those for the representation of the problem P) for ACS.try($x = a$). *Path-forward space complexity*

of an ACS algorithm is the worst case space complexity of the internal data structures for any incremental sequence with n invocations.

In empirical studies, the number of constraint checks is a standard cost measurement for constraint processing. We define for ACS two types of redundant checks. Given a CIS $m_1, m_2, ..., m_k$ and two present values $a \in D_x$ and $b \in D_y$ at $m_t (2 \leq t \leq k)$, a check $c_{xy}(a, b)$ at m_t is a *negative repeat (positive repeat respectively)* iff 1) $(a, b) \notin c_{xy}$ ($(a, b) \in c_{xy}$ respectively), 2) $c_{xy}(a, b)$ was performed at $m_s (1 \leq s < t)$, and 3) b is present from m_s to m_t.

3 ACS in folklore

Traditionally, ACS is simply taken as an implementation of standard AC algorithms in a search procedure. Let us first consider an algorithm ACS-3 employing AC-3. It is shown in Algorithm 2 where only methods different from those in Algorithm 1 are listed.

Algorithm 2: ACS-3 and ACS-3.1record

```
                    ACS-3
ACS-3.init(P_1)
 1  P ← P_1, initialize the internal data structures of ACS-X
 2  return AC-3(P)
ACS-3.backup(timestamp (x, a))
 3  backup the current domains, following timestamp (x, a)
ACS-3.restore(P, timestamp (x, a))
 4  restore the domains, following timestamp (x, a)
ACS-3.hasSupport(x, a, y)
 5  b ← head
 6  while b ← succ(b, D_y) and b ≠ tail do
        if (a, b) ∈ C_xy then return true
 7  return false
                    ACS-3.1record
ACS-3.1record.init(P_1)
 8  P ← P_1
 9  ∀c_xy ∈ P.C and ∀a ∈ D_x, initialize last (x, a, y)
10  return AC-3.1(P)
ACS-3.1record.backup(timestamp (x, a))
11  ∀c_xy ∈ P, a ∈ D_x, backup last (x, a, y), following timestamp (x, a)
12  backup the current domains, following timestamp (x, a)
ACS-3.1record.restore(timestamp (x, a))
13  restore the data structure last (), following timestamp (x, a)
14  restore the domains, following timestamp (x, a)
ACS-3.1record.hasSupport(x, a, y)
15  b ← last(x, a, y); if b ∈ D_y then return true
16  while b ← succ(b, D_y) and b ≠ tail do
17      if (a, b) ∈ c_xy then { last(x, a, y) ← b; return true }
18  return false
```

Proposition 1 ACS-3 is correct with respect to any CIS. Node-forward and path-forward complexity of ACS-3 are both $O(ed^3)$ while node-forward and path-forward space complexity are $O(ed)$. It can not avoid any positive or negative repeats.

It is well known that variable-based AC-3 can be implemented with space $O(n)$. The same is also true for ACS-3.

We next introduce ACS-3.1record, an algorithm that employs AC-3.1 [Bessiere *et al.*, 2005]. It is listed in Algorithm 2 in which methods that are same as the template ACS methods are omitted. AC-3.1 improves upon AC-3 simply by using a data structure last(x, a, y) to remember the first support of a of x in D_y in the latest revision of c_{xy}. When c_{xy} needs to be revised again, for each value a of x, AC-3.1 starts the search of support of a from last(x, a, y). last(x, a, y) satisfies the following two invariants: *support invariant* — $(a,$ last$(x, a, y)) \in c_{xy}$, and *safety invariant* — there exists no support of a in D_y that comes before last(x, a, y). The function hasSupport() (line 15–18) follows the same way as AC-3.1 to find a support. Note that in restore(), the removed values are restored in the *original* ordering of the domains, which is critical for the correctness of ACS-3.1record.

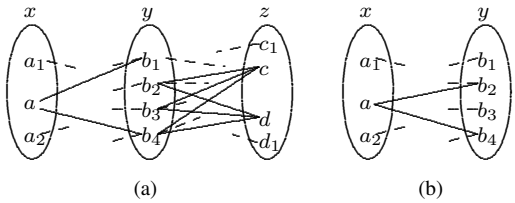

Figure 1: Example

Theorem 1 ACS-3.1record is correct with respect to any CIS. Node-forward and path-forward time complexity of ACS-3.1record are both $O(ed^2)$ while node-forward and path-forward space complexity are $O(ed)$ and $O(ned)$ respectively. It can neither fully avoid negative nor positive repeats.

The node-forward space complexity of ACS-3.1record can be improved to $O(ed \min(n, d))$ [van Dongen, 2003].

Example In this example we focus on how a support is found in a CIS of ACS-3.1record methods. Consider the following CIS: m_i = try$(z = c)$ (returning false) and m_{i+1} = try$(z = d)$. Assume before m_i, P is arc consistent and contains some constraints and domains shown in Figure 1(a) where only the supports of a, c, and d are explicitly drawn. Assume last(x, a, y)=b_1 before m_i. During m_i, we need to find a new support for $a \in D_x$ because b_1 is deleted due to the propagation of $z = c$. Assume last(x, a, y) was updated by ACS-3.1record to the support b_4 before m_i returns false. Since $P \cup \{z = c\}$ is arc inconsistent, last(x, a, y) is restored to be b_1 by m_i. In m_{i+1}, a new support is needed for $a \in D_x$ since b_1 is deleted due to the propagation of $z = d$. ACS-3.1record needs to check b_2 and b_3 before it finds the support b_4. Value b_2 is present from m_i to m_{i+1}, and $(a, b_2) \in c_{xy}$ was checked in both m_i and m_{i+1}. The constraint check $(a, b_2) \in c_{xy}$ is a negative repeat at m_{i+1}.

4 Exploiting residual data

A key feature of ACS-3 and ACS-3.1record is that they are faithful to the respective AC algorithms. We will focus on ACS and investigate new ways to make use of the fact that the methods of an ACS algorithm are usually invoked many times (millions of times to solve a hard problem) by a search procedure.

4.1 ACS-residue

In this section, we design a new algorithm ACS-residue, listed in Algorithm 3, that extends the ideas behind AC-3 and AC-3.1. Like ACS-3.1record, ACS-residue needs a data structure last(x, a, y) for every $c_{xy} \in C$ and $a \in D_x$. After ACS-residue.init(P_1), last(x, a, y) is initialized to be the

first support of a with respect to c_{xy}. At every invocation of ACS-residue.try() or ACS-residue.addInfer(), when finding a support for a value a of D_x with respect to c_{xy}, ACS-residue.hasSupport(x, a, y) first checks (line 3) if last(x, a, y) is still present. If it is, a support is found. Otherwise, it searches (line 3–5) the domain of y from scratch as AC-3 does. If a new support is found, it will be used to update last(x, a, y) (line 5). The method is called ACS-residue because ACS-residue.try() or ACS-residue.addInfer() simply reuses the data left in the last() structure by the previous invocations of try() or addInfer(). Unlike ACS-3.1record, ACS-residue does not maintain last() in backup() and restore().

Algorithm 3: ACS-residue

```
                    ACS-residue
ACS-residue.init(P1) {same as ACS-3.1record.init(P1)}
ACS-residue.backup(timestamp (x, a))
 1  backup the current domains, following timestamp (x, a)
ACS-residue.restore(timestamp (x, a))
 2  restore the domains of P, following timestamp (x, a)
ACS-residue.hasSupport(x, a, y)
 3  if last(x, a, y) ∈ D_y then return true else b ← head
 4  while b ← succ(b, D_y) and b ≠ tail do
 5      if (a, b) ∈ c_xy then { last(x, a, y) ← b ; return true }
 6  return false
                    ACS-resOpt
ACS-resOpt.init(P1)
 7  P ← P1
 8  ∀c_xy ∈ P.C and ∀a ∈ D_x, initialize last (x, a, y) and stop (x, a, y)
 9  return AC-3.1(P)
ACS-resOpt.backup(timestamp (x, a))
10  backup the current domains of P, following timestamp (x, a)
ACS-resOpt.restore(timestamp (x, a))
11  restore the domains, following timestamp (x, a)
ACS-resOpt.try(x = a)
12  ∀c_xy ∈ P.C and ∀a ∈ D_x, stop(x, a, y) ← last(x, a, y)
13  backup(timestamp (x, a))
14  delete all the values except a from D_x, Q ← {(y, x) | c_yx ∈ P.C }
15  if propagate(Q) then return true
16  else {restore(timestamp (x, a)), return false }
ACS-resOpt.addInfer(x ≠ a)
17  ∀c_xy ∈ P.C and ∀a ∈ D_x, stop(x, a, y) ← last(x, a, y)
18  delete a from D_x, Q ← {(y, x) | c_yx ∈ P.C }
19  return propagate(Q)
ACS-resOpt.hasSupport(x, a, y)
20  b ← last(x, a, y) ; if b ∈ D_y then return true
21  while b ← cirSucc(b, D_y^0) and b ∈ D_y and b ≠ stop (x, a, y) do
22      if (a, b) ∈ c_xy then last(x, a, y) ← b; return true
23  return false
```

Theorem 2 ACS-residue is correct with respect to any CIS. Node-forward and path-forward time complexity of ACS-residue are $O(ed^3)$ and $O(ed^3)$ respectively while node-forward and path-forward space complexity are both $O(ed)$. It fully avoids positive repeats but avoids only some negative repeats.

Compared with ACS-3.1record, ACS-residue has a better space complexity but a worse time complexity. ACS-residue does not need to backup its internal data structures.

Example Consider the example in the previous section. Before m_i, i.e., ACS-residue.try($z = c$), the problem is arc consistent and last(x, a, y)=b_1. During m_i, assume ACS-residue updates last(x, a, y) to be b_4 before it returns false. After m_i, only the deleted values are restored to the domains but nothing is done to last() structure and thus last(x, a, y) is still b_4 (b_4 is a residue in last()). In m_{i+1}, i.e., ACS-residue.try($z = d$), when hasSupport() tries to find a support for a of D_x, it checks first last(x, a, y). b_4 is present and thus a support of a. In contrast, ACS-3.1record.try($z = d$) looks for a support for a from $b_1 \in D_y$. Through this example, it is clear that ACS-residue can save some constraint checks that ACS-3.1record can not save (the converse is also true obviously).

4.2 ACS-resOpt

ACS-residue's node-forward complexity is not optimal. We propose another algorithm, ACS-resOpt (listed in Algorithm 3), that has optimal node-forward complexity while using the residues in last(). The idea is to remember the residues in last(x, a, y) by stop(x, a, y) (line 12 and line 17) at the beginning of try() and ACS-resOpt.addInfer(). Then when hasSupport(x, a, y) looks for a support for $a \in D_x$ and last(x, a, y)(=b) is not present, it looks for a new support *after* b (line 21), instead of the beginning of the domain. The search could go through the tail and back to the head and continue until encounter stop(x, a, y). For simplicity, in line 21 of hasSupport(), the initial domain of the problem P is used: cirSucc(a, D_y^0) = succ(head,D_y^0) if succ(a, D_y^0) = tail; otherwise cirSucc(a, D_y^0) = succ(a, D_y^0). In our experiment however, we implement hasSupport() using the *current* domain.

Theorem 3 ACS-resOpt is correct with respect to any CIS. Node-forward and path-forward time complexity are $O(ed^2)$ and $O(ed^3)$ respectively while node-forward and path-forward space complexity are both $O(ed)$. It fully avoids positive repeats but avoids only some negative repeats.

Example Consider the constraint c_{xy} in Figure 1(b) before ACS-resOpt.try(). The supports of a are drawn explicitly in the graph. Assume last(x, a, y) = b_2 before try(). ACS-resOpt.try() will first set stop(x, a, y)=b_2. Assume in the following constraint propagation b_2 is deleted due to other constraints on y. ACS-resOpt.hasSupport(x, a, y) will search a support for a after b_2 and find the new support b_4. Assume b_4 is later deleted by constraint propagation. ACS-resOpt.hasSupport(x, a, y) will start from b_4, go through the tail and back to the head, and finally stop at b_2 because stop(x, a, y)=b_2. No support is found for a and it will be deleted from the current domain.

5 ACS with adaptive domain ordering

To explore the theoretical efficiency limits of ACS, we propose the algorithm ACS-ADO with optimal node-forward and path-forward time complexity. ACS-ADO employs an adaptive domain ordering: a deleted value is simply restored to the end of its domain in restore(), while in ACS-3.1record, it is restored in regard of the total ordering on the initial domain. As a result, when finding a support by using last(), it is sufficient for hasSupport() to search to the end of the domain (rather than going back to the head of the domain as done by ACS-resOpt).

ACS-ADO, listed in Algorithm 4, needs the data structures lastp(x, a, y), buf(a, y) for every $c_{xy} \in P.C$ and $a \in D_x$. The content of lastp(x, a, y) and buf(a, y) is a pointer p to a *supporting node* that has two components $p\uparrow$.bag and $p\uparrow$.for. If buf(a, y)=p, $p\uparrow$.for is a and $p\uparrow$.bag is the set $\{\ b \in D_y \mid$ lastp$(y, b, x)\uparrow$.for=a$\}$. lastp$(x, b, y)\uparrow$.for is a value of D_y.

Algorithm 4: ACS-ADO

```
ACS-ADO.init(P_1)
 1  ∀c_xy ∈ P.C and ∀a ∈ D_x, last(x, a, y) ← head
 2  ∀c_xy ∈ P.C and ∀a ∈ D_x ∪ {tail} lastp(x, a, y) ← NULL
 3  flag ← AC-3.1(P) // AC-3.1 will populate last
 4  foreach c_xy ∈ P.C and each a ∈ D_x do
 5      b ← last(x, a, y)
 6      if buf(b, x) = NULL then buf(b, x) ← createNode(b)
 7      add a to buf(b, x)↑.bag, lastp(x, a, y) ← buf(b, x)
 8  return flag
ACS-ADO.backup(P, timestamp(x, a))
 9  backup the current domains, following timestamp(x, a)
ACS-ADO.restore(P, timestamp(x, a))
10  foreach variable y ∈ P.V do
11      restore the deleted values to the end of D_y, following timestamp(x, a)
12      let v be the first restored value
13      foreach c_xy ∈ P.C do
14          swap buf(v, x) and buf(tail, x)
            swap buf(v, x)↑.for and buf(v,tail)↑.for
ACS-ADO.remove(b, y)
15  c ← succ(b, D_y)
16  if |buf(b, x)↑.bag| > |buf(c, x)↑.bag| then
        swap buf(b, x) and buf(c, x), swap buf(b, x)↑.for and buf(b, x)↑.for
17  foreach v ∈ buf(b, x)↑.bag do update(v, c, x, y, b)
18  delete b from D_y
ACS-ADO.hasSupport(x, a, y)
19  b ← lastp(x, a, y)↑.for
20  if (a, b) ∈ c_xy then return true else b_1 ← b
21  while b ← succ(b, D_y) and b ≠ tail do
22      if (a, b) ∈ c_xy then {update(a, b, x, y, b_1), return true }
23  update(a, tail, x, y, b_1), return false
ACS-ADO.createNode(b)
24  create a supporting node p such that p↑.bag ← {}, p↑.for ← b
25  return p
ACS-ADO.update(a, b, x, y, b_1)
26  delete a from buf(b_1, x)↑.bag
27  if buf(b, x) = NULL then buf(b, x) ← createNode(b)
28  add a to buf(b, x)↑.bag, lastp(x, a, y) ← buf(b, x)
```

ACS-ADO maintains on lastp(x, a, y) the *safety invariant*, i.e., there is no support of a before the value lastp$(x, a, y)\uparrow$.for in D_y, and the *presence invariant*, i.e., lastp$(x, a, y)\uparrow$.for is present in D_y or the tail of D_y. It also maintains the *correspondence invariant*, i.e., lastp$(x, a, y)\uparrow$.for=b if and only if $a \in$ buf$(b, x)\uparrow$.

With the safety and presence invariants on lastp(), to find a support of $a \in D_x$ with respect to c_{xy}, ACS-ADO.hasSupport(x, a, y) starts from lastp$(x, a, y)\uparrow$.for(line 19) and stops by the tail of D_y (line 21). When a new support is found, lastp(x, a, y) is updated (line 22) by `update` that guarantees the correspondence invariant (line 26–28). ACS-ADO.hasSupport assures the safety invariant on lastp. When removing a value, say b of D_y, ACS-ADO.remove(b, y) finds the first present value c after b (line 15) and makes buf(b, x) point to the node with smaller bag (line 16). It then updates (line 17) lastp(x, a, y) for all $a \in$ buf$(b, x)\uparrow$.bag. In this way, we always update the lastp() structures for a smaller number of values. When restoring deleted values, for each variable y, ACS-ADO.restore(timestamp(x, a)) restores all the deleted values after timestamp(x, a) to the end of D_y (line 11). Note that tail is greater than any values in D_y. Since there might be some values whose lastp$(x, a, y)\uparrow$.for is tail of D_y, we need to swap the supporting nodes in buf(v, x) and buf(tail, x) (line 13–14).

Example Figure 2(a) shows the data structures of the values of D_x with respect to c_{xy}. The nodes with 1 to 4 and a to d represent values of D_y and D_x. The value of a node disconnected from the linked list is *not* in the current domain. The nodes with area labelled by "bag" are supporting nodes.

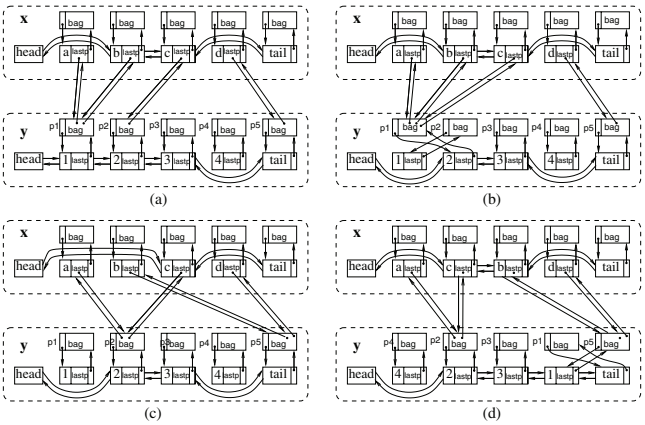

Figure 2: Examples for remove() and restore()

The arrow from a supporting node, say p_1, to the value node 1 means $p_1\uparrow$.for=1, and the arrow from a value node, say 1, to the supporting node p_1 implies buf$(1, x)$=p_1. An arrow from the lastp area of a value node, say a, to a supporting node p_1 implies lastp$(x, a, y)\uparrow$.for=p_1. An arrow from the bag area of a supporting node, say p_1, to a value node, say a, implies that $a \in p_1\uparrow$.bag. Note that the details of lastp (buf respectively) structures of the values of D_y (D_x respectively) are omitted.

Assume value 1 is removed. Since buf$(1, x)\uparrow$.bag is larger than that of value 2 that is the first present successor of 1, the method remove() swaps buf$(1, x)$ and buf$(2, x)$ and swap buf$(1, x)\uparrow$.for and buf$(2, x)\uparrow$.for. Now a and b are pointing to value 2 through p_1. Then c of $p_2\uparrow$.bag is made to point p_1. Figure 2(b) shows structures after the removal of 1.

Consider another data structures shown in Figure 2(c). Assume 1 needs to be restored to D_y. Since 1 is the first restored value, all values pointing to tail should now point to 1. The method restore() simply swaps buf$(1, x)$ and buf(tail, x) and swaps buf$(1, x)\uparrow$.for and buf(tail,x)$\uparrow$.for (Figure 2(d)). In constant time, all the values previously pointing to tail are now pointing to 1 through the supporting node p_5. □

Proposition 2 Consider any incremental sequence with n invocations, The cumulated worst-case time complexity of ACS-ADO.remove() in the sequence is $O(ed \lg d)$.

Theorem 4 ACS-ADO is correct with respect to any CIS. Node-forward and path-forward time complexity are $O(ed^2)$ while node-forward and path-forward space complexity are $O(ed)$. ACS-ADO fully avoids negative repeats but does not avoid any positive repeats.

6 Experiments

The new ACS algorithms are benchmarked on random binary constraint problems and Radio Link Frequency Assignment Problems (RLFAPs). The sample results on problems ($n = 50, e = 125$) at phase transition area are shown in Figure 3 where the constraint checks are the average of 50 instances and the time is total amount for those instances. The experiments were carried out on a DELL PowerEdge 1850 (two 3.6GHz Intel Xeon CPUs) with Linux 2.4.21. We use dom/deg variable ordering and lexicographical value ordering. The constraint check in our experiment is very cheap.

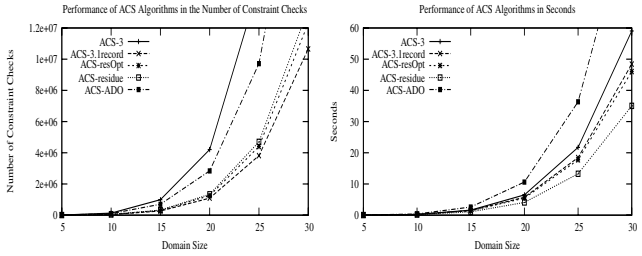

Figure 3: Experiments on random problems

In terms of constraint checks, ACS-3 and ACS-ADO are significantly worse than the rest. However, the difference among ACS-3.1record, ACS-resOpt, and ACS-residue is marginal. This shows the considerable savings led by the reuse of residual data given the fact that the node-forward complexity of ACS-residue is not optimal. However, ACS-resOpt is only slightly better than AC-residue although it improves the node-forward complexity of the latter to be optimal. ACS-ADO has the best theoretical time complexity among the new algorithms, but it has the worst experimental performance. One explanation is that it loses the benefit of residual support due to that fact that $\mathsf{lastp}(x, a, y)\uparrow.\mathsf{for}$ is guaranteed to be present but might not be a support of a. In summary, the use of residues bring much more savings than other complexity improvements of the new algorithms.

Since conducting roughly the same amount of constraint checks, the great simplicity of ACS-residue makes it a clear winner over other algorithm in terms of clock time. The performance of ACS3-resOpt is very close to that of AC-3.1record. Since ACS-ADO uses a rather heavy data structure, its running time becomes the worst.

The above observations also hold on RLFAP problems (see the table below). (ACS-resOpt uses less number of checks than ACS-3.1record but is the slowest because it conducts more domain checks than the others.)

RLFAP	ACS-3	ACS-3.1record	ACS-resOpt	ACS-residue	ACS-ADO
scen11	124.5M	22.7M	20.8M	23.1M	85.6M
	3.265s	3.394s	4.961s	1.934s	3.456s

7 Related works and conclusions

We are not aware of any other work that has made a clear difference between standalone AC and AC during search. However, there does exist a number of works that began to look at specific algorithms to enforce AC in the context of a search procedure. Lecoutre and Hemery (2006) confirmed the effectiveness of ACS-residue in random problems and several other real life problems and extend it by multidirectionality of constraints. The work by Regin (2005) focuses specifically on reducing the space cost to embed an AC-6 based algorithm to MAC while keeping its complexity the same as that of standalone optimal AC algorithms on any branch of a search tree. However, experimental data has not been published for this algorithm.

In this paper we propose to study ACS as a whole rather than just an embedding of AC algorithm into a search procedure. Some complexity measurements are also proposed to distinguish new ACS algorithms (e.g., ACS-residue and ACS-resOpt) although an implementation of ACS using optimal AC algorithm with backup/restore mechanism, e.g., ACS-3.1record, can always achieve the best node and path time complexity.

A new perspective brought by ACS is that we have more data to (re)use and we do not have to follow exactly AC algorithms when designing ACS algorithms. For example, ACS-residue uses ideas from both AC-3 and AC-3.1 but also shows clear difference from either of them. The simplicity and efficiency of ACS-residue vs. other theoretically more efficient algorithms reminds of a situation that occurred around 1990 when it was found AC-3 empirically outperformed AC-4 [Wallace, 1993] but finally the ideas behind them led to better AC algorithms. We expect that the effort on improving theoretical complexity of ACS algorithms will eventually contribute to empirically more efficient algorithms.

The success of ACS-residue and our extensive empirical study suggest a few directions for the study of ACS algorithms. 1) We need extensive theoretical and empirical study on possible combinations of the new ACS algorithms and traditional filtering algorithms including AC-6/7. We have combined for example residue and ADO, which significantly improved the performance of ADO. 2) We notice in our experiments that the optimal complexity of ACS-3.1record does not show significant gain over ACS-residue even in terms of the number of constraint checks while efforts to improve ACS-residue (e.g., ACS-resOpt) gain very little. This phenomenon is worth of studying to boost the performance of ACS algorithms. 3) The ACS perspective provides fresh opportunities to reconsider the numerous existing filtering algorithms (including singleton arc consistency and global constraints) in the context of search.

8 Acknowledgements

This material is based in part upon works supported by the Science Foundation Ireland under Grant 00/PI.1/C075 and by NASA under Grant NASA-NNG05GP48G.

References

[Bessiere *et al.*, 2005] C. Bessiere, J.C. Regin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Lecoutre and Hemery, 2006] Christophe Lecoutre and Fred Hemery. A study of residual supports in arc consistency. Manuscript, 2006.

[Mackworth, 1977] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):118–126, 1977.

[Régin, 2005] Jean-Charles Régin. Maintaining arc consistency algorithms during the search without additional space cost. In *Proceedings of CP-05*, pages 520–533, 2005.

[Sabin and Freuder, 1994] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of PPCP-94*, pages 10–20, 1994.

[van Dongen, 2003] M. R. C. van Dongen. To avoid repeating checks does not always save time. In *Proceedings of AICS'2003*, Dublin, Ireland, 2003.

[Wallace, 1993] Richard J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings of IJCAI-93*, pages 239–247, Chambery, France, 1993.