Refining the Basic Constraint Propagation Algorithm[1]

Christian Bessire[2]
LIRMM–CNRS (UMR 5506)
161 rue Ada
34392 Montpellier Cedex 5, France
`bessiere@lirmm.fr`

Jean-Charles Rgin
ILOG
1681, route des Dolines
06560 Valbonne, France
`regin@ilog.fr`

Abstract

Propagating constraints is the main feature of any constraint solver. This is thus of prime importance to manage constraint propagation as efficiently as possible, justifying the use of the best algorithms. But the ease of integration is also one of the concerns when implementing an algorithm in a constraint solver. This paper focuses on AC-3, which is the simplest arc consistency algorithm known so far. We propose two refinements that preserve as much as possible the ease of integration into a solver (no heavy data structure to be maintained during search), while giving some noticeable improvements in efficiency. One of the proposed refinements is analytically compared to AC-6, showing interesting properties, such as optimality of its worst-case time complexity.

1 Introduction

Constraint propagation is the basic operation in constraint programming. It is now well-recognized that its extensive use is necessary when we want to efficiently solve hard constraint satisfaction problems. All the constraint solvers use it as a basic step. Thus, each improvement that can be incorporated in a constraint propagation algorithm has an immediate effect on the behavior of the constraint solving engine. In practical applications, many constraints are of well-known types for which specific algorithms are available. These algorithms generally receive a set of removed values for one of the variables involved in the constraint, and propagate these deletions according to the constraint. They are usually as cheap as one can expect in cpu time. This state of things implies that most of the existing solving engines are based on a constraint-oriented or variable-oriented propagation scheme (ILOG Solver, CHOCO, etc.). And AC-3, with its natural both constraint-oriented [Mackworth, 1977] and variable-oriented [McGregor, 1979] propagation of the constraints, is the generic constraint propagation algorithm which fits the best this propagation scheme. Its successors, AC-4, AC-6, and AC-7, indeed, were written with a value-oriented propagation. This is one of the reasons why AC-3 is the algorithm which is usually used to propagate those constraints for which nothing special is known about the semantics (and then for which no specific algorithm is available). This algorithm has a second strong advantage when compared to AC-4, AC-6 or AC-7, namely, its independence with regard to specific data structure which should be maintained if used during a search procedure. (And following that, a greater easiness to implement it.) On the contrary, its successors, while more efficient when applied to networks where much propagation occurs ([Bessière *et al.*, 1995; Bessire *et al.*, 1999]), need to maintain some additional data structures.

In this paper, our purpose is to present two new algorithms, `AC2000` and `AC2001`, which, like AC-3, accept variable-oriented and constraint-oriented propagation, and which improve AC-3 in efficiency (both in terms of constraint checks and cpu time). `AC2000`, like AC-3, is free of any data structure to be maintained during search. `AC2001`, at the price of a slight extra data structure (just an integer for each value-constraint pair) reaches an optimal worst-case time complexity.[3] It leads to substantial gains, which are shown both on randomly generated and real-world instances of problems. A comparison with AC-6 shows interesting theoretical properties. Regarding the human cost of their implementation, `AC2000` needs a few lines more than the classical AC-3, and `AC2001` needs the management of its additional data structure.

2 Preliminaries

Constraint network. A finite binary *constraint network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is defined as a set of n *variables* $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$, where $D(X_i)$ is the finite set of possible *values* for variable X_i, and a set $\mathcal{C}$ of e binary *constraints* between pairs of variables. A constraint C_{ij} on the ordered set of variables (X_i, X_j) is a subset of the Cartesian product $D(X_i) \times D(X_j)$ that specifies the *allowed* combinations of values for the variables X_i and X_j. (For each constraint C_{ij}, a constraint C_{ji} is defined between (X_j, X_i), allowing the same pairs of values

[1]This work has been partially financed by ILOG under a research collaboration contract ILOG/CNRS/University of Montpellier II.

[2]Member of the COCONUT group.

[3]A related paper by Zhang and Yap appears in these proceedings.

in the reverse order.) Verifying whether a given pair (v_i, v_j) is allowed by C_{ij} or not is called a *constraint check*. A *solution* of a constraint network is an instantiation of the variables such that all the constraints are satisfied.

Arc consistency. Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network, and C_{ij} a constraint in $\mathcal{C}$. A value $v_i \in D(X_i)$ is *consistent with* C_{ij} iff $\exists v_j \in D(X_j)$ such that $(v_i, v_j) \in C_{ij}$. (v_j is then called a *support* for (X_i, v_i) on C_{ij}.) A value $v_i \in D(X_i)$ is *viable* iff it has support in all $D(X_j)$ such that $C_{ij} \in \mathcal{C}$. $\mathcal{P}$ is *arc consistent* iff all the values in all the domains are viable. We achieve arc consistency in $\mathcal{P}$ by removing every value which is not viable.

3 A first stab at improving AC-3

3.1 Background on AC-3

Before presenting the algorithm we propose in this section, let us briefly recall the AC-3 algorithm. We present it with the structure proposed by McGregor [McGregor, 1979], which is built on a variable-based propagation scheme. This will be recommended for the algorithm presented in the next subsection. The main algorithm (see Algorithm 1) is very close to the original AC-3, with an initialization phase (lines 1 to 3), a propagation phase (line 4), and with the use of the function `Revise3`(X_i, X_j) that removes from $D(X_i)$ the values without support in $D(X_j)$ (line 2). But instead of handling a queue of the constraints to be propagated, it uses a queue[4] of the variables that have seen their domain modified. When a variable X_j is picked from the queue (line 2 of `Propagation3` in Algorithm 2), all the constraints involving X_j are propagated with `Revise3`. This is the only change w.r.t. the Mackworth's version of AC-3.[5] (This algorithm has been presented in [Chmeiss and Jgou, 1998] under the name AC-8.)

Algorithm 1: `Main algorithm`

function `AC` (in $\mathcal{X}$: set)**: Boolean**
1 $Q \leftarrow \emptyset$;
  for each $X_i \in X$ **do**
    for each X_j *such that* $C_{ij} \in \mathcal{C}$ **do**
2   **if** `Revise-X`$(X_i, X_j,$ ***false***[6]$)$ **then**
        if $D(X_i) = \emptyset$ **then** return **false** ;
3       $Q \leftarrow Q \cup \{X_i\}$;
4 return `Propagation-X`(Q);

3.2 The algorithm `AC2000`

If we closely examine the behavior of AC-3, we see that removing a single value v_j from a domain $D(X_j)$ (inside function `Revise3`) is enough to put X_j in the propagation queue (line 3 of `AC` in Algorithm 1 and line 5 of `Propagation3` in Algorithm 2), and to provoke a call to `Revise3`(X_i, X_j) for every constraint C_{ij} involving X_j (lines 3 and 4 of `Propagation3`). `Revise3` will look for a support for every value in $D(X_i)$ whereas for some of them v_j was perhaps not even a support. (As a simple example, we can take the constraint $X_i = X_j$, where $D(X_i) = D(X_j) = [1..11]$. Removing value 11 from $D(X_j)$ leads to a call to `Revise3`(X_i, X_j), which will look for a support for every value in $D(X_i)$, for a total cost of $1+2+\ldots+9+10+10 = 65$ constraint checks, whereas only $(X_i, 11)$ had lost support.) We exploit this remark in `AC2000`. Instead of looking blindly for a support for a value $v_i \in D(X_i)$ each time $D(X_j)$ is modified, we do that only under some conditions. In addition to the queue Q of variables modified, we use a second data structure, $\Delta(X_j)$,[7] which for each variable X_j contains the values removed since the last propagation of X_j. When a call to `Revise2000`$(X_i, X_j, lazymode)$ is performed, instead of systematically looking whether a value v_i still has support in $D(X_j)$, we first check that v_i really lost a support, namely one of its supports is in $\Delta(X_j)$ (line 3 of `Revise2000` in Algorithm 3). The larger $\Delta(X_j)$ is, the more expensive that process is, and the greater the probability to actually find a support to v_i in this set is. So, we perform this "lazymode" only if $\Delta(X_j)$ is sufficiently smaller than $D(X_j)$. We use a parameter Ratio to decide that. (See line 1 of `Propagation2000` in Algorithm 3.) The Boolean lazymode is set to true when the ratio is not reached. Otherwise, lazymode is false, and `Revise2000` performs exactly as `Revise3`, going directly to lines 4 to 5 of `Revise2000` without testing the second part of line 3.

If we run `AC2000` on our previous example, we have $\Delta(X_j) = \{11\}$. If $|\Delta(X_j)| < \mathsf{Ratio} \cdot |D(X_j)|$, then for each $v_i \in D(X_i)$, we check whether $\Delta(X_j)$ contains a support of v_i before looking for a support for v_i. This requires 11 constraint checks. The only value for which support is effectively sought is $(X_i, 11)$. That requires 10 additional

Algorithm 2: `Subprocedures for AC-3`

function `Propagation3` (in Q: set)**: Boolean**
1 **while** $Q \neq \emptyset$ **do**
2   pick X_j from Q;
3   **for each** X_i *such that* $C_{ij} \in \mathcal{C}$ **do**
4   **if** `Revise-X`(X_i, X_j) **then**
        if $D(X_i) = \emptyset$ **then** return **false** ;
5       $Q \leftarrow Q \cup \{X_i\}$;
  return **true** ;

function `Revise3` (in X_i, X_j: variable)**: Boolean**
  $\mathsf{CHANGE} \leftarrow$ **false**;
  for each $v_i \in D(X_i)$ **do**
    if $\nexists v_j \in D(X_j) / C_{ij}(v_i, v_j)$ **then**
      remove v_i from $D(X_i)$;
      $\mathsf{CHANGE} \leftarrow$ **true**;
  return CHANGE ;

[4]We name it a queue, but, as in AC-3, it has to be implemented as a set since in line 3 of Algorithm 1 and in line 5 of Algorithm 2 we add X_i to Q only if it does not already belong to it.

[5]In fact, McGregor's version of the function `Revise` differs from Mackworth's one. Algorithm `Revise3` is the Mackworth's version [Mackworth, 1977].

[6]The third parameter is useless for AC-3 but will be used in the next version of `Revise`.

[7]We take this name from [Van Hentenryck *et al.*, 1992], where Δ denotes the same thing, but for a different use.

Algorithm 3: `Subprocedures for AC2000`

function **`Propagation2000`** **(in** Q: set**): Boolean**

  while $Q \neq \emptyset$ **do**
    pick X_j from Q;
1   lazymode$\leftarrow$ $(|\Delta(X_j)| < \mathsf{Ratio} \cdot |D(X_j)|)$;
    for each X_i *such that* $C_{ij} \in \mathcal{C}$ **do**
      if $\mathtt{Revise2000}(X_i, X_j, lazymode)$ **then**
        if $D(X_i) = \emptyset$ **then** return **false** ;
        $Q \leftarrow Q \cup \{X_i\}$;
2   reset $\Delta(X_j)$;
  return **true** ;

function **`Revise2000`** **(in** X_i, X_j: variable;
                in lazymode: **Boolean): Boolean**

  CHANGE $\leftarrow$ **false**;
  for each $v_i \in D(X_i)$ **do**
3   **if** $\neg lazymode$ ***or*** $\exists v_j \in \Delta(X_j) / C_{ij}(v_i, v_j)$ **then**
4   **if** $\nexists v_j \in D(X_j) / C_{ij}(v_i, v_j)$ **then**
        remove v_i from $D(X_i)$;
        add v_i to $\Delta(X_i)$;
5   CHANGE $\leftarrow$ **true**;
  return CHANGE ;

constraint checks. We save 44 constraint checks compared to AC-3.

Analysis

Let us first briefly prove `AC2000` correctness. Assuming AC-3 is correct, we just have to prove that the lazy mode does not let arc-inconsistent values in the domain. The only way the search for support for a value v_i in $D(X_i)$ is skipped, is when we could not find a support for v_i in $\Delta(X_j)$ (line 3 of `Revise2000`). Since $\Delta(X_j)$ contains all the values deleted from $D(X_j)$ since its last propagation (line 2 of `Propagation2000`), this means that v_i has exactly the same set of supports as before on C_{ij}. Thus, looking again for a support for v_i is useless. It remains consistent with C_{ij}.

The space complexity of `AC2000` is bounded above by the sizes of Q and Δ. Q is in $O(n)$ and Δ is in $O(nd)$, where d is the size of the largest domain. This gives a $O(nd)$ overall complexity. In this space complexity, it is assumed that we built `AC2000` with the variable-oriented propagation scheme, as recommended earlier. If we implement `AC2000` with the constraint-oriented propagation scheme of the original AC-3, we need to attach a $\Delta(X_j)$ to each constraint C_{ij} put in the queue. This implies a $O(ed)$ space complexity.

The organization of `AC2000` is the same as in AC-3. The main change is in function `Revise2000`, where $\Delta(X_j)$ and $D(X_j)$ are examined instead of only $D(X_j)$. Their total size is bounded above by d. This leads to a worst-case where d^2 checks are performed, as in `Revise3`. Thus, the overall time complexity is in $O(ed^3)$ since `Revise2000` can be called d times per constraint. This is as in AC-3.

4 `AC2001`

In Section 3, we proposed an algorithm, which, like AC-3, does not need special data structures to be maintained during search. (Except the current domains, which have to be maintained by any search algorithm performing some look-ahead.) During a call to `Revise2000`, for each v_i in $D(X_i)$, we have to look whether v_i has a support in $\Delta(X_j)$, to know whether it lost supports or not. In this last case, we have to look again for a support for v_i on C_{ij} in the whole $D(X_j)$ set. If we could remember what was the support found for v_i in $D(X_j)$ the last time we revised C_{ij}, the gain would be twofold: First, we would just have to check whether this last support has been removed from $D(X_j)$ or not, instead of exploring the set $\Delta(X_j)$. Second, when a support was effectively removed from $D(X_j)$, we would just have to explore the values in $D(X_j)$ that are "after" that last support since "predecessors" have already been checked before. Adding a very light extra data structure to remember the last support of a value on a constraint leads to the algorithm `AC2001` that we present in this section.

Let us store in $Last(X_i, v_i, X_j)$ the value that has been found as a support for v_i at the last call to `Revise2001`(X_i, X_j). The function `Revise2001` will always run in the lazy mode since the cost of checking whether the $Last$ support on C_{ij} of a value v_i has been removed from $D(X_j)$ is not dependent on the number of values removed from $D(X_j)$. A second change w.r.t. `AC2000` is that the structure Δ is no longer necessary since the test "$Last(X_i, v_i, X_j) \notin D(X_j)$" can replace the test "$Last(X_i, v_i, X_j) \in \Delta(X_j)$". The consequence is that `AC2001` can equally be used with a constraint-based or variable-based propagation. We only present the function `Revise2001`, which simply replaces the function `Revise3` in AC-3. The propagation procedure is that of AC-3, and the $Last$ structure has to be initialized to `NIL` at the beginning. In line 1 of `Revise2001` (see Algorithm 4) we check whether $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$. If it is not the case, we look for a new support for v_i in $D(X_j)$; otherwise nothing is done since $Last(X_i, v_i, X_j)$ is still in $D(X_j)$. In line 2 of `Revise2001` we can see the second advantage of storing $Last(X_i, v_i, X_j)$: If the supports are checked in $D(X_j)$ in a given ordering "$<_d$", we know that there isn't any support for v_i before $Last(X_i, v_i, X_j)$ in $D(X_j)$. Thus, we can look for a new support only on the values greater than $Last(X_i, v_i, X_j)$.

Algorithm 4: `Subprocedure for AC2001`

function **`Revise2001`** **(in** X_i, X_j: variable**): Boolean**

  CHANGE $\leftarrow$ **false**;
  for each $v_i \in D(X_i)$ **do**
1   **if** $Last(X_i, v_i, X_j) \notin D(X_j)$ **then**
2   **if** $\exists v_j \in D(X_j) / v_j >_d Last(X_i, v_i, X_j) \wedge C_{ij}(v_i, v_j)$
      then $Last(X_i, v_i, X_j) \leftarrow v_j$;
      else
        remove v_i from $D(X_i)$;
        CHANGE $\leftarrow$ **true**;
  return CHANGE ;

On the example of Section 3, when $(X_j, 11)$ is removed, `AC2001` simply checks for each $v_i \in [1..10]$ that $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$, and finds that $Last(X_i, 11, X_j)$ has been removed. Looking for a new support for 11 does not need any constraint check since $D(X_j)$

		AC-3		AC2000		AC2001		*AC-6*$^{(*)}$
		#ccks	time	#ccks	time	#ccks	time	*time*
$\langle 150, 50, 500, 1250 \rangle$	(under-constrained)	100,010	0.04	100,010	0.05	100,010	0.05	*0.07*
$\langle 150, 50, 500, 2350 \rangle$	(over-constrained)	507,783	0.18	507,327	0.18	487,029	0.16	*0.10*
$\langle 150, 50, 500, 2296 \rangle$	(phase transition)	2,860,542	1.06	1,601,732	0.69	688,606	0.34	*0.32*
$\langle 50, 50, 1225, 2188 \rangle$	(phase transition)	4,925,403	1.78	3,038,280	1.25	1,147,084	0.61	*0.66*
SCEN#08	(arc inconsistent)	4,084,987	1.67	3,919,078	1.65	2,721,100	1.25	*0.51*

Table 1: Arc consistency results in mean number of constraint checks (#ccks) and mean cpu time in seconds (time) on a PC Pentium II 300MHz (50 instances generated for each random class). (*) The number of constraint checks performed by AC-6 is similar to that of AC2001, as discussed in Section 6.

does not contain any value greater than $Last(X_i, 11, X_j)$, which was equal to 11. It saves 65 constraint checks compared to AC-3.

Analysis

Proving correctness of AC2001 can be done very quickly since the framework of the algorithm is very close to AC-3. They have exactly the same initialization phase except that AC2001 stores $Last(X_i, v_i, X_j)$, the support found for each v_i on each C_{ij}. (In line 1 it is assumed that NIL does not belong to $D(X_j)$.) During the propagation, they diverge in the way they revise an arc. As opposed to AC-3, Revise2001(X_i, X_j) goes into a search for support for a value v_i in $D(X_i)$ only if $Last(X_i, v_i, X_j)$ does not belong to $D(X_j)$. We see that checking that $Last(X_i, v_i, X_j)$ still belongs to $D(X_j)$ is sufficient to ensure that v_i still has a support in $D(X_j)$. And if a search for a new support has to be done, limiting this search to the values of $D(X_j)$ greater than $Last(X_i, v_i, X_j)$ w.r.t. to the ordering $<_d$ used to visit $D(X_j)$ is sufficient. Indeed, the previous call to Revise2001 stopped as soon as the value $Last(X_i, v_i, X_j)$ was found. It was then the smallest support for v_i in $D(X_j)$ w.r.t. $<_d$.

The space complexity of AC2001 is bounded above by the size of Q, and $Last$. Q is in $O(n)$ or $O(e)$, depending on the propagation scheme that is used (variable-based or constraint-based). $Last$ is in $O(ed)$ since each value v_i has a $Last$ pointer for each constraint involving X_i. This gives a $O(ed)$ overall complexity.

As in AC-3 and AC2000, the function Revise2001 can be called d times per constraint in AC2001. But, at each call to Revise2001(X_i, X_j), for each value $v_i \in D(X_i)$, there will be a test on the $Last(X_i, v_i, X_j)$, and a search for support only on the values of $D(X_j)$ greater than $Last(X_i, v_i, X_j)$. Thus, the total work that can be performed for a value v_i over the d possible calls to Revise2001 on a pair (X_i, X_j) is bounded above by d tests on $Last(X_i, v_i, X_j)$ and d constraint checks. The overall time complexity is then bounded above by $d \cdot (d + d) \cdot 2 \cdot e$, which is in $O(ed^2)$. This is optimal [Mohr and Henderson, 1986]. AC2001 is the first optimal arc consistency algorithm proposed in the literature that is free of any lists of supported values. Indeed, the other optimal algorithms, AC-4, AC-6, AC-7, and AC-Inference all use these lists.

5 Experiments

In the sections above, we presented two refinements of AC-3, namely AC2000 and AC2001. It remains to see whether they are effective in saving constraint checks and/or cpu time when compared to AC-3. AS we said previously, the goal is not to compete with AC-6/AC-7, which have very subtle data structure for the propagation phase. An improvement (even small) w.r.t. AC-3 would fulfill our expectations. However, we give AC-6 performances, just as a marker.

5.1 Arc consistency as a preprocessing

The first set of experiments we performed should permit to see the effect of our refinements when arc consistency is used as a preprocessing (without search). In this case, the chance to have some propagations is very small on real instances. We have to fall in the phase transition of arc consistency (see [Gent *et al.*, 1997]). So, we present results for randomly generated instances (those presented in [Bessire *et al.*, 1999]), and for only one real-world instance. For the random instances, we used a model B generator [Prosser, 1996]. The parameters are $\langle N, D, C/p1, T/p2 \rangle$, where N is the number of variables, D the size of the domains, C the number of constraints (their density $p1 = 2C/N \cdot (N-1)$), and T the number of forbidden tuples (their tightness $p2 = T/D^2$). The real-world instance, SCEN#08, is taken from the FullRLFAP archive,[8] which contains instances of radio link frequency assignment problems (RLFAPs). They are described in [Cabon *et al.*, 1999]. The parameter Ratio used in AC2000 is set to 0.2. Table 1 presents the results for four classes of random instances plus the real-world one.

The upper two are under-constrained ($\langle 150, 50, 500/0.045, 1250/\mathbf{0.5} \rangle$) and over-constrained ($\langle 150, 50, 500/0.045, 2350/\mathbf{0.94} \rangle$) problems. They represent cases where there is little or no propagation to reach the arc consistent or arc inconsistent state. This is the best case for AC-3, which performs poorly during propagation. We can see that AC2000 and AC2001 do not suffer from this.

The third and fourth experiments are at the phase transition of arc consistency for sparse ($\langle 150, 50, 500/\mathbf{0.045}, 2296/0.918 \rangle$) and dense ($\langle 50, 50, 1225/\mathbf{1.0}, 2188/0.875 \rangle$) problems. We can assume there is much propagation on these problems before reaching the arc consistent state. This has a significant impact on the respective efficiencies of the algorithms. The smarter

[8] We thank the Centre d'Electronique de l'Armement (France).

	MAC-3		MAC2000		MAC2001		*MAC6*
	#ccks	time	#ccks	time	#ccks	time	*time*
SCEN#01	5,026,208	2.33	4,319,423	2.10	1,983,332	1.62	*2.05*
SCEN#11	77,885,671	39.50	77,431,840	38.22	9,369,298	21.96	*14.69*
GRAPH#09	6,269,218	2.95	5,164,692	2.57	2,127,598	1.99	*2.41*
GRAPH#10	6,790,702	3.04	5,711,923	2.65	2,430,109	1.85	*2.17*
GRAPH#14	5,503,326	2.53	4,253,845	2.13	1,840,886	1.66	*1.90*

Table 2: Results for search of the first solution with a MAC algorithm in mean number of constraint checks (#ccks) and mean cpu time in seconds (time) on a PC Pentium II 300MHz.

the algorithm is, the lower the number of constraint checks is. AC2001 dominates AC2000, which itself dominates AC-3. And the cpu time follows this trend.

The lower experiment reports the results for the SCEN#08. This is one of the instances in FullRLFAP for which arc consistency is sufficient to detect inconsistency.

5.2 Maintaining arc consistency during search

The second set of experiments we present in this section shows the effect of our refinements when arc consistency is maintained during search (MAC algorithm [Sabin and Freuder, 1994]) to find the first solution. We present results for all the instances contained in the FullRLFAP archive for which more than 2 seconds were necessary to find a solution or prove that none exists. We took again 0.2 for the Ratio in AC2000. It has to be noticed that the original question in these instances is not satisfiability but the search of the "best" solution, following some criteria. It is of course out of the scope of this paper.

From these instances we can see a slight gain for AC2000 on AC-3. On SCEN#11, it can be noticed that with a smaller Ratio, AC2000 slightly improves its performances. (Ratio = 0.05 seems to be the best.) A more significant gain can be seen for AC2001, with up to 9 times less constraint checks and twice less cpu time on SCEN#11. As for the experiments performed on random instances at the phase transition of arc consistency, this tends to show that the trick of storing the $Last$ data structure significantly pays off. However, we have to keep in mind that we are only comparing algorithms with simple data structures. This prevents them from reaching the efficiency of algorithms using lists of supported values when the amount of propagation is high, namely on hard problems. (E.g., a MAC algorithm using AC-6 for enforcing arc consistency needs only 14.69 seconds to solve the SCEN#11 instance.)

6 AC2001 vs AC-6

In the previous sections, we proposed two algorithms based on AC-3 to achieve arc consistency on a binary constraint network. AC2000 is close to AC-3, from which it inherits its $O(ed^3)$ time complexity and its $O(nd)$ space complexity. AC2001, thanks to its additional data structure, has an optimal $O(ed^2)$ worst-case time complexity, and an $O(ed)$ space complexity. These are the same characteristics as AC-6.[9] So, we can ask the question: "What are the differences between AC2001 and AC-6?".

Let us first briefly recall the AC-6 behavior [Bessire, 1994]. AC-6 looks for one support (the *first* one or *smallest* one with respect to the ordering $<_d$) for each value (X_i, v_i) on each constraint C_{ij} to prove that (X_i, v_i) is currently viable. When (X_j, v_j) is found as the smallest support for (X_i, v_i) on C_{ij}, (X_i, v_i) is added to $S[X_j, v_j]$, the list of values currently having (X_j, v_j) as smallest support. If (X_j, v_j) is removed from $D(X_j)$, it is added to the $DeletionSet$, which is the stream driving propagations in AC-6. When (X_j, v_j) is picked from the $DeletionSet$, AC-6 looks for the *next* support (i.e., greater than v_j) in $D(X_j)$ for each value (X_i, v_i) in $S[X_j, v_j]$. Notice that the $DeletionSet$ corresponds to $\sum_{X_j \in \mathcal{X}} \Delta(X_j)$ in AC2000, namely the set of values removed but not yet propagated.

To allow a closer comparison, we will suppose in the following that the $S[X_i, v_i]$ lists of AC-6 are split on each constraint C_{ij} involving X_i, leading to a structure $S[X_i, v_i, X_j]$, as in AC-7.

Property 1 *Let $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ be a constraint network. If we suppose AC2001 and AC-6 follow the same ordering of variables and values when looking for supports and propagating deletions, then, enforcing arc consistency on $\mathcal{P}$ with AC2001 requires the same constraint checks as with AC-6.*

Proof. Since they follow the same ordering, both algorithms perform the same constraint checks in the initialization phase: they stop search for support for a value v_i on C_{ij} as soon as the first v_j in $D(X_j)$ compatible with v_i is found, or when $D(X_j)$ is exhausted (then removing v_i). During the propagation phase, both algorithms look for a new support for a value v_i on C_{ij} only when v_i has lost its current support v_j in $D(X_j)$ (i.e., $v_i \in S[X_j, v_j, X_i]$ for AC-6, and $v_j = Last(X_i, v_i, X_j)$ for AC2001). Both algorithms start the search for a new support for v_i at the value in $D(X_j)$ immediately greater than v_j w.r.t. the $D(X_j)$ ordering. Thus, they will find the same new support for v_i on C_{ij}, or will remove v_i, at the same time, and with the same constraint checks. And so on. □

From property 1, we see that the difference between AC2001 and AC-6 cannot be characterized by the number of constraint checks they perform. We will then focus on the way they find which values should look for a new support. For that, both algorithms handle their specific data structures. Let us characterize the number of times each of them checks

[9] We do not speak about AC-7 here, since it is the only one among these algorithms to deal with the bidirectionality of the constraints (namely, the fact that $C_{ij}(v_i, v_j) = C_{ji}(v_j, v_i)$).

its own data structure when a set $\Delta(X_j)$ of deletions is propagated on a given constraint C_{ij}.

Property 2 *Let C_{ij} be a constraint in a network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. Let $\Delta(X_j)$ be a set of values removed from $D(X_j)$ that have to be propagated on C_{ij}. If,*
$A = |\Delta(X_j)| + \sum_{v_j \in \Delta(X_j)} |S[X_j, v_j, X_i]|$,
$B = |D(X_i)|$, and
C = # checks performed on C_{ij} to propagate $\Delta(X_j)$,
then, $A + C$ and $B + C$ represent the number of operations AC-6 and `AC2001` *will respectively perform to propagate $\Delta(X_j)$ on C_{ij}.*

Proof. From property 1 we know that AC-6 and `AC2001` perform the same constraint checks. The difference is in the process leading to them. AC-6 traverses the $S[X_j, v_j, X_i]$ list for each $v_j \in \Delta(X_j)$ (i.e., A operations), and `AC2001` checks whether $Last(X_i, v_i, X_j)$ belongs to $D(X_j)$ for every v_i in $D(X_i)$ (i.e., B operations). □

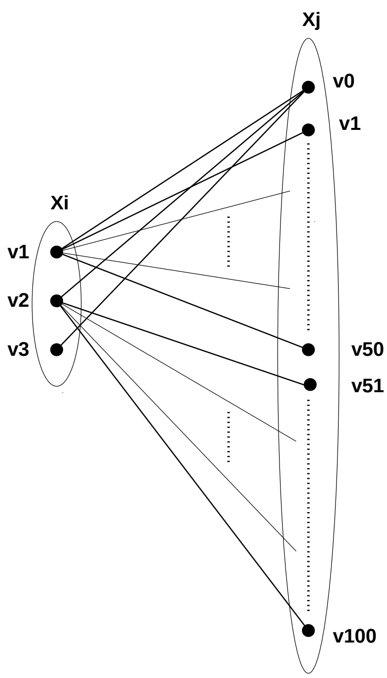

Figure 1: The constraint example

We illustrate this on the extreme case presented in Figure 1. In that example, the three values of X_i are all compatible with the first value v_0 of X_j. In addition, (X_i, v_1) is compatible with all the values of X_j from v_1 to v_{50}, and (X_i, v_2) with all the values of X_j from v_{51} to v_{100}. Imagine that for some reason, the value v_3 has been removed from $D(X_i)$ (i.e., $\Delta(X_i) = \{v_3\}$). This leads to $A = 1$, $B = 101$, and $C = 0$, which is a case in which propagating with AC-6 is much better than with `AC2001`, even if none of them needs any constraint check. Indeed, AC-6 just checks that $S[X_i, v_3, X_j]$ is empty,[10] and stops. `AC2001` takes one by one the 101 values v_j of $D(X_j)$ to check that their $Last(X_j, v_j, X_i)$ is not in $\Delta(X_i)$. Imagine now that the values v_1 to v_{100} of $D(X_j)$ have been removed (i.e., $\Delta(X_j) = \{v_1, \ldots, v_{100}\}$). Now, $A = 100$, $B = 3$, and $C = 0$. This means that `AC2001` will clearly outperform AC-6. Indeed, AC-6 will check for all the 100 values v_j in $\Delta(X_j)$ that $S[X_j, v_j, X_i]$ is empty,[11] while `AC2001` just checks that $Last(X_i, v_i, X_j)$ is not in $\Delta(X_j)$ for the 3 values in $D(X_i)$.

[10]The only value compatible with (X_i, v_3) is (X_j, v_0), which is currently supported by (X_i, v_1).

[11]Indeed, (X_j, v_0) is the current support for the three values in $D(X_i)$ since it is the smallest in $D(X_j)$ and it is compatible with every value in $D(X_i)$.

Discussion

Thanks to property 2, we have characterized the amount of effort necessary to AC-6 and `AC2001` to propagate a set $\Delta(X_j)$ of removed values on a constraint C_{ij}. $A - B$ gives us information on which algorithm is the best to propagate $\Delta(X_j)$ on C_{ij}. We can then easily imagine a new algorithm, which would start with an AC-6 behavior on all the constraints, and would switch to `AC2001` on a constraint C_{ij} as soon as $A - B$ would be positive on this constraint (and then forget the S lists on C_{ij}). Switching from `AC2001` to AC-6 is no longer possible on this constraint because we can deduce in constant time that $Last(X_i, v_i, X_j) = v_j$ when v_i belongs to $S[X_j, v_j, X_i]$, but we cannot obtain cheaply $S[X_j, v_j, X_i]$ from the $Last$ structure. A more elaborated version would maintain the S lists even in the `AC2001` behavior (putting v_i in $S[X_j, v_j, X_i]$ each time v_j is found as being the $Last(X_i, v_i, X_j)$). This would permit to switch from AC-6 to `AC2001` or the reverse at any time on any constraint in the process of achieving arc consistency. These algorithms are of course far from our initial purpose of emphasizing easiness of implementation since they require the S and $Last$ structures to be maintained during search.

7 Non-binary versions

Both `AC2000` and `AC2001` can be extended to deal with non-binary constraints. A support is now a tuple instead of a value. Tuples in a constraint $C(X_{i_1}, \ldots, X_{i_q})$ are ordered w.r.t. the ordering $<_d$ of the domains, combined with the ordering of $X_{i_1}, \ldots, X_{i_q}$ (or any order used when searching for support). Once this ordering is defined, a call to `Revise2000`(X_i, C) —because of a set $\Delta(X_j)$ of values removed from $D(X_j)$— simply checks for each $v_i \in D(X_i)$ whether there exists a support τ of v_i on the constraint C for which $\tau[X_j]$ (the value of X_j in the tuple) belongs to $\Delta(X_j)$. If yes, it looks for a new support for v_i on C. `Revise2001`(X_i, C) checks for each $v_i \in D(X_i)$ whether $Last(X_i, v_i, C)$, which is a tuple, still belongs to $D(X_{i_1}) \times \cdots \times D(X_{i_q})$ before looking for a new support for v_i on C.

This extension to non-binary constraints is very simple to implement. However, it has to be handled with care when the variable-oriented propagation is used, as recommended for `AC2000`. (With a constraint-based propagation, a $\Delta(X_j)$ set is duplicated for each constraint put in the queue to propagate it.) Variable-based propagation is indeed less precise in the way it drives propagation than constraint-based propagation. Take the constraint $C(X_{i_1}, X_{i_2}, X_{i_3})$ as an example. If $D(X_{i_1})$ and $D(X_{i_2})$ are modified consecutively, X_{i_1} and X_{i_2} are put in the queue Q consecutively. Picking X_{i_1} from Q implies the calls to $\texttt{Revise}(X_{i_2}, C)$ and $\texttt{Revise}(X_{i_3}, C)$, and picking X_{i_2} implies the calls to $\texttt{Revise}(X_{i_1}, C)$ and $\texttt{Revise}(X_{i_3}, C)$. We see that $\texttt{Revise}(X_{i_3}, C)$ is called twice while once was enough. To overcome this weakness, we need to be more precise in the way we propagate deletions. The solution, while being technically simple, is more or less dependent on the architecture of the solver in which it is used. Standard techniques are described in [ILOG, 1998; Laburthe, 2000].

8 Conclusion

We presented AC2000 and AC2001, two refinements in AC-3. The first one improves slightly AC-3 in efficiency (number of constraint checks and cpu time) although it does not need any new data structure to be maintained during search. The second, AC2001, needs an additional data structure, the $Last$ supports, which should be maintained during search. This data structure permits a significant improvement on AC-3, and decreases the worst-case time complexity to the optimal $O(ed^2)$. AC2001 is the first algorithm in the literature achieving optimally arc consistency while being free of any lists of supported values. Its behavior is compared to that of AC-6, making a contribution to the understanding of the different AC algorithms, and opening an opportunity of improvement. This is in the same vein as the work on AC-3 vs AC-4 [Wallace, 1993], which was leading up to AC-6.

Acknowledgements

We would like to thank Philippe Charman who pointed out to us the negative side of value-oriented propagation. The first author also wants to thank all the members of the OCRE team for the discussions we had about the specification of the CHOCO language.

References

[Bessière *et al.*, 1995] C. Bessière, E. C. Freuder, and J. C. Rgin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[Bessire *et al.*, 1999] C. Bessire, E.C. Freuder, and J.C. Rgin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[Bessire, 1994] C. Bessire. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[Cabon *et al.*, 1999] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4:79–89, 1999.

[Chmeiss and Jgou, 1998] A. Chmeiss and P. Jgou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[Gent *et al.*, 1997] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings CP'97*, pages 327–340, Linz, Austria, 1997.

[ILOG, 1998] ILOG. *User's manual*. ILOG Solver, 4.3 edition, 1998.

[Laburthe, 2000] F. Laburthe. *User's manual*. CHOCO, 0.39 edition, 2000.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[McGregor, 1979] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[Mohr and Henderson, 1986] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Prosser, 1996] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA, 1994.

[Van Hentenryck *et al.*, 1992] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wallace, 1993] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings IJCAI'93*, pages 239–245, Chambry, France, 1993.