1

Using bidirectionality to speed up arc-consistency processing

Christian Bessière[1]
Jean-Charles Régin[2]

ABSTRACT In [Bes94a, BC93], Bessière and Cordier said that the AC-6 arc-consistency algorithm is optimal in time on constraint networks where nothing is known about the constraint semantics. However, in constraint networks, it is always assumed that constraints are bidirectional. None of the previous algorithms achieving arc-consistency (AC-3 [Mac77, MF85], AC-4 [MH86], AC-6) use constraint bidirectionality. We propose here an improved version of AC-6 which uses this property. Then, we claim that our new algorithm is optimal in the number of constraint checks performed (i.e. given a variable, value, and arc ordering, it performs the minimum possible number of constraint checks according to these orders).

1.1 Introduction

In the last five years, the number of applications using constraint networks has dramatically increased. It appears that the more constraint networks are used, the simpler the constraint satisfaction techniques involved in the applications are. In fact, a great part of real-life applications using constraint networks are limited to a forward-checking search procedure [HE80], or use an arc-consistency filtering algorithm before or during the search. This is one of the reasons why arc-consistency remains a hot area in the CSP community [Bes94a, BC93, Per92, VDT92]. Improving the efficiency of arc-consistency algorithms improves in the same way the efficiency of all applications using arc-consistency as a filtering step.

Recently, Bessière and Cordier [Bes94a, BC93] provided an algorithm, AC-6, which they claim to be optimal in time on constraint networks where nothing is known about the constraint semantics. However, in constraint networks, it is always assumed that constraints are bidirectional (i.e. if the

[1]LIRMM, UMR 9928 University of Montpellier II/CNRS, 161, rue Ada, 34392 Montpellier Cedex 5, FRANCE e-mail : bessiere@lirmm.fr

[2]GDR 1093 CNRS, LIRMM, UMR 9928 University of Montpellier II/CNRS, 161, rue Ada, 34392 Montpellier Cedex 5, FRANCE e-mail : regin@lirmm.fr

combination of values a for a variable i and b for a variable j is allowed by the constraint between i and j, then the combination b for j and a for i is allowed by the constraint between j and i), and none of the previous algorithms (AC-3 [Mac77, MF85], AC-4 [MH86], and AC-6) use this fact[3].

In this paper, we propose AC6++, an improved version of AC-6 which uses constraint bidirectionality. We claim that AC6++ is optimal in the number of constraint checks performed[4]. Moreover, AC6++ avoids the explosion of space complexity, keeping it at $O(ed)$, like AC-6 (where e is the number of constraints and d the size of the largest domain).

The rest of the paper is organized as follows. Section 1.2 gives some preliminary definitions on constraint networks and arc-consistency. Section 1.3 presents the algorithm AC6++. In section 1.4, an experimental comparison of AC-6 and AC6++ is given. Section 1.5 contains a conclusion.

1.2 Background

A *network of binary constraints* (CN) is defined as a set of n variables $\{i, j, ...\}$, a domain $D = \{D_i, D_j, ...\}$ where D_i is the finite set of possible values for variable i, and a set of binary constraints between variables. A *binary constraint* (or relation) R_{ij} between variables i and j is a subset of the Cartesian product $D_i \times D_j$ that specifies the allowed pairs of values for i and j. $R_{ij}(a, b)$ is true when the pair consisting of the element a of D_i and the element b of D_j is permitted; value false means the pair is not permitted. In all networks of interest here constraints are *bidirectional*, i.e. $R_{ij}(a, b) = R_{ji}(b, a)$. This is not a restriction.

A *directed graph* G can be associated to a constraint network, where nodes correspond to variables in the CN and two arcs (i, j) and (j, i) link nodes i and j every time there is a relation R_{ij} (and so R_{ji}) on variables i and j in the CN. In the following, the function Γ will be used, where for every node i, $\Gamma(i)$ is the set of nodes j such that $(i, j) \in arcs(G)$.

A solution of a constraint network is an instantiation of the variables such that all constraints are satisfied.

Having the constraint R_{ij}, value b in D_j is called a *support* for value a in D_i if the pair (a, b) is allowed by R_{ij}. A value a for a variable i is *viable* if for every variable j such that R_{ij} exists, a has a support in D_j. The domain D of a CN is *arc-consistent* for this CN if for every variable i in the CN, all values in D_i are viable. When a CN has a domain D, we call

[3] E.C. Freuder has just proposed an algorithm, AC-7, which uses that fact. AC6++ and AC-7 have been first presented in the same workshop ([BR94] and [Fre94] respectively)

[4] An improved version of AC-6 (named AC6+), which uses constraint bidirectionality, was proposed in [Bes94b]. But it was not optimal in the number of constraint checks performed.

maximal arc-consistent domain of this CN domain $AC(D)$, defined as the union of all domains included in D and arc-consistent for this CN. $AC(D)$ is also arc-consistent and is the domain expected to be computed by an arc-consistency algorithm.

1.3 The new algorithm using bidirectionality

1.3.1 Preamble

As we have seen in the definition above, and as Mohr and Henderson underlined in [MH86], arc-consistency is based on the notion of support. As long as a value a for a variable i (denoted by (i, a)) has supporting values on each of the other variables j linked to i in the constraint graph, a is considered a viable value for i. But once there is a variable on which no remaining value satisfies the relation with (i, a), then a must be eliminated from D_i.

AC-6, which assigns an ordering of the values in every D_i, checks one support (the first one) for each value (i, a) on each constraint R_{ij} to prove that (i, a) is currently viable. When (j, b) is found as the smallest support of (i, a) on R_{ij}, (i, a) is added to S_{jb}, the list of values currently having (j, b) as smallest support. If (j, b) is removed from D_j then AC-6 looks for the next support in D_j for each value (i, a) in S_{jb}. But AC-6 does not use the fact that $R_{ij}(a, b) = R_{ji}(b, a)$. When it looks for a support for (j, b) on R_{ji}, it does not use the fact that $R_{ij}(a, b)$ has perhaps already been checked for some (i, a) when looking for supports for values in D_i.

The purpose of AC6++ is to achieve arc-consistency after an optimal number of constraint checks, using constraint bidirectionality. It only performs a constraint check $R_{ij}(a, b)$ when none of the previously done constraint checks is sufficient to prove that (i, a) has a support in D_j, and moreover, when neither $R_{ij}(a, b)$ nor $R_{ji}(b, a)$ have been checked before. It is built to maintain true the properties below during the whole algorithm:

1. never checking $R_{ij}(a, b)$ if there exists b' still in D_j such that $R_{ij}(a, b')$ has already been successfully checked.
2. never checking $R_{ij}(a, b)$ if there exists b' still in D_j such that $R_{ji}(b', a)$ has already been successfully checked.
3. never checking $R_{ij}(a, b)$ if:
   (a) it has already been checked
   (b) $R_{ji}(b, a)$ has been checked before.
4. keeping an $O(ed)$ space complexity.

Properties (1), (2), and (3) define what we call the optimality of AC6++. So, optimality is defined according to a variable ordering, a value ordering, and an arc ordering. It does not imply the minimum number of constraint checks, but a minimal number according to the ordering used [WF92].

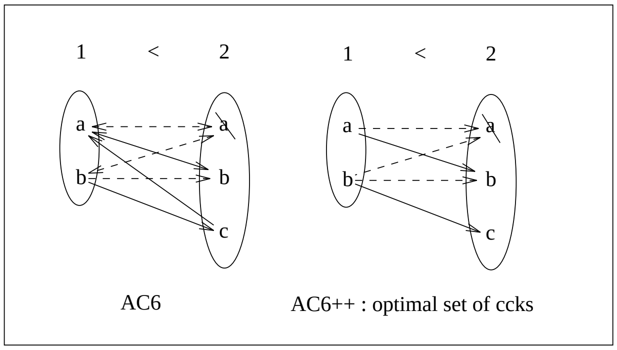

FIGURE 1.1. A sample CSP

Consider the simple problem of Fig. 1.1 for illustration purposes. There are two variables *1* and *2* with domains $\{a, b\}$ and $\{a, b, c\}$ respectively, linked by the constraint "inferior in the lexicographic order". An arrow from a value v for *1* to a value w for *2* indicates that the constraint check $R_{12}(v, w)$ has been performed. A double ended arrow indicates that both $R_{12}(v, w)$ and $R_{21}(w, v)$ have been checked. A solid arrow indicates a successful check; a dotted arrow indicates that the check was false. In this example, AC-6 needs 9 constraint checks where only 5 were necessary to achieve arc-consistency. AC6++ avoids useless constraint checks by enforcing properties (1), (2), and (3). Property (2) forbids $R_{21}(b, a)$ and $R_{21}(c, a)$. Property (3b) forbids $R_{21}(a, a)$ and $R_{21}(a, b)$ in addition. Both of these properties forbid $R_{21}(b, a)$[5]. Properties (1) and (3a) were already enforced by AC-6 (but not by AC-3 or AC-4).

Every time a constraint check $R_{ji}(b, a)$ is performed, AC6++ needs to store this information to avoid future $R_{ij}(a, b)$ or $R_{ij}(a, b')$ checks forbidden by properties (3b) and (2). This storage must be done in a careful way to avoid falling into the trap of an $O(ed^2)$ space complexity. Then, there are two differences with the data structure of AC-6. First, S_{ia} sets are split into S_{ija} sets: each value (i, a) has a set of values supported by it per constraint instead of one for all the constraints. $b \in S_{ija}$ implies (i, a) is known as supporting (j, b) on R_{ji}, then (i, a) is compatible with (j, b). Hence, an $R_{ij}(a, b)$ is checked if and only if $S_{ija} = \emptyset$, otherwise we know (i, a) has a support on R_{ij} without any new constraint check. Property (2) holds. Property (3b) holds for successful constraint checks. Second, arrays *inf*-

[5] We can see that neither $R_{12}(a, c)$ nor $R_{21}(c, a)$ are checked by AC6++.

support are added. *inf-support*$[(i,j),a]$ represents the smallest value in D_j which may be compatible with (i,a). These arrays are needed to ensure property (3b) for negative constraint checks and property (3a)[6]. Property (1) is inherent in the principle of AC-6.

1.3.2 The algorithm

Here is the data structure of AC6++:

- A table M of Booleans keeps track of which values of the initial domain are in the current domain or not ($M(i,a)=\textbf{true} \Leftrightarrow a \in D_i$). In this table, each initial D_i is considered as the integer range $1..|D_i|$. We use the following constant time functions and procedures to handle D_i lists:
  - *last*(D_i) returns the greatest value in D_i if $D_i \neq \emptyset$, else returns 0.
  - if $a \in D_i \backslash$*last*(D_i), *next*(a, D_i) returns the smallest value in D_i greater than a.
  - *remove*(a, D_i) removes the value a from D_i. *remove*(a, D_i) can be extended to stop the arc-consistency algorithm if a was the only remaining value in D_i. The CN, indeed, has no solution.
- For all a in D_i, S_{ija} contains all values b in D_j for which (i,a) is assigned as its current support. The *current* support is not necessary the smallest one, as opposed to AC-6.
- In AC6++, we need to record arrays *inf-support*$[(i,j),a]$ which were not needed in AC-6. These arrays are updated by AC6++ to ensure that every b in D_j compatible with (i,a) is greater than or equal to *inf-support*$[(i,j),a]$.
- *Waiting-List* (as in AC-4 or AC-6) contains values deleted from the domain but for which the propagation of the deletion has not yet been processed.

The function *nextsupport* (see Fig. 1.2), which looks for the smallest value in D_j not smaller than b and supporting (i,a) on R_{ij}, is the same as in AC-6 except the addition of the test of line 3 to avoid checking $R_{ij}(a,b)$ when $R_{ji}(b,a)$ has already been checked as false (i.e. when we know that (j,b) has no support in D_i until a value greater than a).

---

[6] Property (3a) held in AC-6 without *inf-support* arrays because we knew that (i,a) was always the smallest support for values in S_{ia}. In AC6++ we lose this strong property.

function nextsupport(**in** i, j, a: integer; **in out** b: integer) : **Boolean**

/* returns true iff a value b is found */

1 **if** $b >$ last(D_j) **then** *return* **false**;
2 **while not** $M(j, b)$ **do** $b \leftarrow b + 1$;
3 **if** *inf-support*$[(j, i), b] \leq a$ **then**
4   **if** $R_{ij}(a, b)$ **then** *return* **true**;
5 **if** $b <$ last(D_j) **then**
    $b \leftarrow$ next(b, D_j);
    goto line 3;
6 *return* **false**;

FIGURE 1.2.

Another function, *find-support* (see Fig. 1.3), is used in AC6++ to check whether an S_{ija} set contains values in D_j or not. Every time *find-support* unsuccessfully checks if a value b in S_{ija} is in D_j, b is deleted from S_{ija} to avoid checking it again in the following.

function find-support(**in out** S_{ija} : set; **out** b : *integer*) : **Boolean**

/* returns true iff a value b is found */

1 **if** $S_{ija} = \emptyset$ **then** *return* **false**;
2 $b \leftarrow$ an element of S_{ija};
3 **if** $M(j, b)$ **then** *return* **true**;
4 delete b from S_{ija};
5 **goto** line 1;

FIGURE 1.3.

The algorithm AC6++ roughly looks like AC-6, with an initialization step and a propagation step (see Fig. 1.4). The initialization step looks for at least one support for each value (i, a) on each constraint R_{ij}. But the search of support for (i, a) on R_{ij} is only done if S_{ija} does not contain values in D_j; otherwise, this search is useless: (i, a) has supports on R_{ij} (since values supported by (i, a) are supports for (i, a)). If the search of support is done for a value (i, a) on R_{ij} and a value b is found, a is added to S_{jib} since b is now the current support of (i, a). *inf-support*$[(i, j), a]$ records b as the smallest value in D_j supporting (i, a) on R_{ij} since the search follows the order of D_j.

The propagation step takes values (j, b) from the *Waiting-List* and propagates the consequences of their deletion. It looks for another support for values a in S_{jib} sets if they still belong to D_i and if they have no other known support on R_{ij} ($S_{ija} \cap D_j = \emptyset$).

```
{initialization}
Waiting-List ← Empty-List;
for (i, a) ∈ 𝒟 do M(i, a) ← true;
for (i, j) ∈ arcs(G) do
    for a ∈ D_i do
        S_ija ← ∅;
        inf-support[(i, j), a] ← 1; /* first value of D_j */

for (i, j) ∈ arcs(G) do
    for a ∈ D_i do
        if find-support(S_ija, b) /* b is out */ then
            put a in S_jib
        else
            b ← inf-support[(i, j), a];
            if nextsupport(i, j, a, b) /* b is in out */ then
                put a in S_jib;
                inf-support[(i, j), a] ← b
            else
                remove(a, D_i); M(i, a) ← false;
                Add-to(Waiting-List, (i, a));

{propagation}
while Waiting-List ≠ ∅ do
    pick (j, b) from Waiting-List;
    for i ∈ Γ(j) do
        for a ∈ S_jib do
            delete a from S_jib;
            if M(i, a) then
                if find-support(S_ija, c) /* c is out */ then
                    put a in S_jic
                else
                    c ← inf-support[(i, j), a];
                    if nextsupport(i, j, a, c) /* c is in out */ then
                        put a in S_jic;
                        inf-support[(i, j), a] ← c
                    else
                        remove(a, D_i); M(i, a) ← false;
                        Add-to(Waiting-List, (i, a));
```

FIGURE 1.4. AC6++

During the whole algorithm, every time a value is found without support on a constraint, it is removed from D and put in the *Waiting-List*.

In the example, after the processing of arc *(1,2)*, the data structure of AC6++ is in the following state: $S_{12v} = \emptyset,\ \forall v \in D_1; S_{21a} = \emptyset; S_{21b} = \{a\}; S_{21c} = \{b\}$; *inf-support*$[(1,2),a] = b$; *inf-support*$[(1,2),b] = c$; *inf-support*$[(2,1),v] = a,\ \forall v \in D_2$. After the processing of arc *(2,1)* some changes appear: $S_{12a} = \{b\}; S_{12b} = \{c\}$; a is removed from $D_2; \forall v \in D_2$, *inf-support*$[(2,1),v]$ is unchanged.

1.3.3 Proof of correctness of AC6++

AC6++ is similar to AC-6 except that it does not always ensure to know the smallest support for a value on a constraint, but only the existence of a support and an inferior bound below which no support exists. After the initialization step, the following property holds until the end of the algorithm:

$$(\alpha)\ \forall (i,a) \in D, \forall R_{ij}, \forall (j,b) \in D : R_{ij}(a,b) \Rightarrow b \geq \textit{inf-support}[(i,j),a]$$

Hence, when (i,a) has no current support, and no checked supports in D_j ($S_{ija} \cap D_j = \emptyset$), we can start looking for another one at *inf-support*$[(i,j),a]$ without forgetting any support of (i,a). Moreover, we do not need to check $R_{ij}(a,b)$ for values b in D_j such that *inf-support*$[(j,i),b] > a$ because of property (α). So, despite line 3 of the function *nextsupport*, an (i,a) support cannot be forgotten. Thus, value (i,a) is only removed from D_i (lines 17 and 33) when it has no support in D_j on a constraint R_{ij}. If all previously removed values are out of $AC(D^0)$ (D^0 being the initial domain), then (i,a) is out of $AC(D^0)$. $AC(D^0)$ is trivially included in D when AC6++ is started (since $D^0 = D$). Then, by induction, (i,a) is out of $AC(D^0)$. Thus, $AC(D^0) \subseteq D$ is an invariable property of AC6++.

Every time a value (j,b) is removed, it is put in the *Waiting-List* until the values it was supporting are checked for other supports. Every time a value (i,a) is found without support on a constraint, it is removed from D. Thus, every value (i,a) in D has at least one support in $D \cup$ *Waiting-List* on each constraint R_{ij}. AC6++ terminates with an empty *Waiting-List*. Hence, after AC6++, every value in D has a support in D on each constraint. Thus, D is arc-consistent.

$AC(D^0) \subseteq D$ and D arc-consistent at the end of AC6++ implies that D is the maximal arc-consistent domain $AC(D^0)$ at the end of AC6++.

1.3.4 Proof of optimality

In this section, we will show that the properties (1), (2), (3a), and (3b) of section 1.3.1 are kept true by AC6++.

With the same principle as AC-6, properties (1) and (3a) hold. Indeed, we stop looking for supports for a value (i,a) on R_{ij} as soon as we find the first support, and an $R_{ij}(a,b)$ cannot be checked twice since we start looking for a new support for (i,a) on R_{ij} where we stopped the previous time.

Every time a constraint check $R_{ji}(b,a)$ is successfully performed (when looking for a support for (j,b)), AC6++ records b in S_{ija}. Since AC6++ only looks for a support for (i,a) on R_{ij} if $S_{ija} \cap D_j = \emptyset$, we are sure that $R_{ij}(a,b)$ is only checked when none of the previously done constraint checks on R_{ji} can prove that (i,a) has a support in D_j. Property (2) holds.

Finally, we have to prove that when we check $R_{ij}(a,b)$, $R_{ji}(b,a)$ has never been checked before. If $R_{ji}(b,a)$ has already been successfully checked, b must be in S_{ija} or out of D, and $R_{ij}(a,b)$ cannot be checked. If $R_{ji}(b,a)$ has been checked as false, that means that we have looked for a support for (j,b) in D_i and that (i,a) was not a support. So, if $b \in D_j$, *inf-support*$[(j,i),b]$ is greater than a, and line 3 of function *nextsupport* avoids checking $R_{ij}(a,b)$. Property (3b) holds.

1.3.5 Time and space complexity

The two inner loops of AC6++ are in *nextsupport* and *find-support*. Since each $R_{ij}(a,b)$ is checked at most once, in *nextsupport*, complexity due to the calls to this function is bounded above by $O(ed^2)$, as in AC-6. During the whole algorithm, a value $a \in D_i$ is put at most once in S_{jib}: when (j,b) is assigned as the current support of (i,a). *find-support*, which contains the second inner loop of AC6++, is called at most d times for each S_{ija} (when a value b current support of (i,a) is removed from D_j), for a total work of at most d deletions in each S_{ija}. *find-support* performs $O(d)$ on each S_{ija}, and so, complexity due to the calls to this function is bounded above by $2ed \times O(d)$ i.e. $O(ed^2)$. Thus, $O(ed^2)$ is the worst-case time complexity for AC6++, as for AC-6.

The worst-case space complexity of AC-6 was $O(ed)$ because of the size of S_{jb} sets. In AC6++, S_{jb} sets of AC-6 are split into S_{jib} sets. This does not increase their total size: each arc-value pair $[(i,j),a]$ has at most one current support (j,b) (i.e. a belonging to S_{jib}). Hence, the total size of the S_{jib} sets is bounded above by $O(ed)$.

Since all supports already found for a value are recorded (to avoid useless checks), this is not a trivial result, the intuition leading us to expect $O(ed^2)$. This result is based on the fact that in AC6++, as in AC-6, during the whole algorithm, the number of $R_{ij}(a,b)$ successfully checked with (i,a) and (j,b) still in $D \cup$ *Waiting-List* is bounded above by $O(ed)$. We have to add arrays *inf-support*, which take a $2ed$ space. Thus, the total space complexity of AC6++ remains $O(ed)$.

1.4 Experimental results

We tested the performances of the two algorithms on the same problems as in [Bes94a]. For each problem, we counted the number of constraint checks performed by each algorithm, as opposed to [Bes94a]. Moreover, the two algorithms use the extension of the procedure *remove* defined in 1.3.2 (i.e. exit from the algorithm as soon as a domain D_i is empty, which is called domain "wipe-out" by Wallace [Wal93]).

The first comparison was the zebra problem [Dec90]. We obtained the following results:

AC-6	:	1025	constraint checks
AC6++	:	734	constraint checks

In Fig. 1.5, we then compared the two algorithms on the n-queens problem (i.e. a $n \times n$ chessboard on which we want to put n queens, none of them attacked by any other).

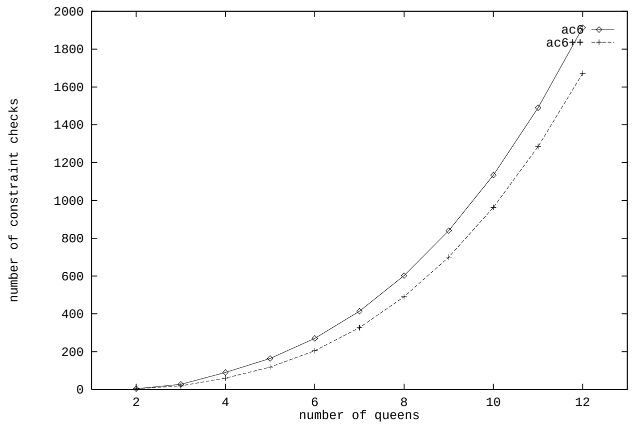

FIGURE 1.5. Comparison of AC-6 and AC6++ on the n-queens problem

Finally, we compared the two algorithms on the classes of randomly generated constraint networks used in [Bes94a]. Four parameters were taken into account: n the number of variables, d the number of values per variable, pc the probability that a constraint R_{ij} between two variables exists, and pu the probability in existing relations R_{ij} that a pair of values $R_{ij}(a, b)$ is allowed. The result given for each class is the average of thirty instances of problems in the class so as to be more representative of the class. In Fig. 1.6, 1.7, and 1.8, a vertical line shows the borderline between problems where wipe-out is generally produced (located on the left of the line), and problems where arc-consistency is produced (on the right).

In average, on the tested problems, AC6++ is around 20% better than AC-6 in the number of constraint checks performed. The interesting remark we made is that in these 20%, $R_{ij}(a, b)$ checks avoided because $R_{ji}(b, a)$ has already been checked only represent about 10%. The other 10% are $R_{ij}(a, b)$ checks avoided because there exists $b \neq b'$ such that $R_{ji}(b', a)$ has already been successfully checked.

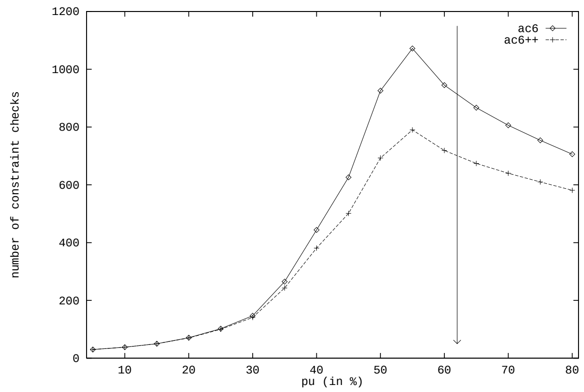

FIGURE 1.6. AC-6 and AC6++ on randomly generated CNs with 20 variables having 5 possible values, where the probability pc to have a constraint between two variables is 30%

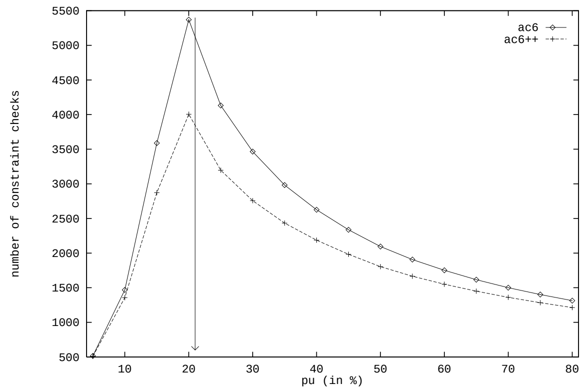

FIGURE 1.7. AC-6 and AC6++ on randomly generated CNs with 12 variables having 16 possible values, where the probability pc to have a constraint between two variables is 50%

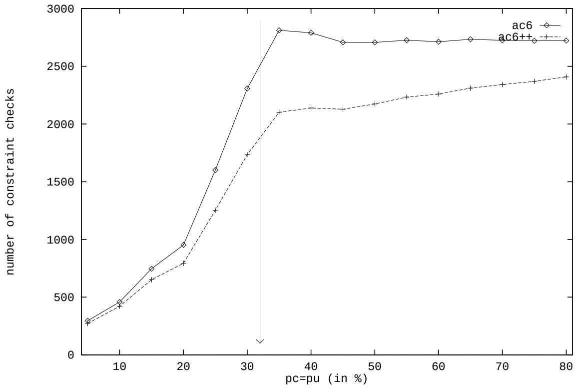

FIGURE 1.8. AC-6 and AC6++ on randomly generated CNs with 18 variables having 9 possible values

1.5 Conclusion

We have provided an algorithm, AC6++, which achieves arc-consistency in binary constraint networks where constraints are bidirectional. It uses this property to avoid useless constraint checks. It performs a minimal number of constraint checks w.r.t. the variable, value, and arc ordering, without increasing the $O(ed)$ worst-case space complexity of AC-6. Its worst-case time complexity is $O(ed^2)$ as for AC-6. Experimental comparison shows that it significantly outperforms AC-6 in the number of constraint checks performed.

Acknowledgements: The work of J.C. Régin was supported by SANOFI-CHIMIE. Moreover, we want to thank Christophe FIORIO for his LaTeX algorithm style.

1.6 References

[BC93] C. Bessière and M.O. Cordier. Arc-consistency and arc-consistency again. In *AAAI-93, Proceedings Eleventh National Conference on Artificial Intelligence*, pages 108–113, Washington, DC, 1993.

[Bes94a] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.

[Bes94b] C. Bessière. A fast algorithm to establish arc-consistency in constraint networks. Technical Report TR 94-003, LIRMM University of Montpellier II, France, January 1994.

[BR94] C. Bessière and J.C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *ECAI'94, Proceedings of the Workshop on Constraint Processing*, pages 9–16, Amsterdam, The Netherlands, 1994.

[Dec90] R. Dechter. Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[Fre94] E.C. Freuder. Using metalevel constraint knowledge to reduce constraint checking. In *ECAI'94, Proceedings of the Workshop on Constraint Processing*, pages 27–33, Amsterdam, The Netherlands, 1994.

[HE80] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[MF85] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[MH86] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[Per92] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[VDT92] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[Wal93] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *IJCAI'93, Proceedings Thirteenth International Joint Conference on Artificial Intelligence*, pages 239–245, Chambéry, France, 1993.

[WF92] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.