Improving Arc-Consistency Algorithms with Double-Support Checks

M.R.C. van Dongen and J.A. Bowen
Computer Science Department
University College Cork, Ireland

August, 2000

Abstract

Arc-consistency algorithms are widely used to simplify Constraint Satisfaction Problems. The new notion of a *double-support check* is presented to improve the average performance of arc-consistency algorithms. The improvement is that, where possible, consistency-checks are used to find supports for *two* values, one value in the domain of each variable, which were previously known to be unsupported. It is motivated by the insight that *in order to minimize the number of consistency-checks it is necessary to maximize the number of uncertainties which are resolved per check.* The idea is used to improve AC-3 and DEE and results in a new general purpose arc-consistency algorithm called AC-3_b. Experimental results of a comparison of AC-3, DEE, AC-3_b and AC-7 are presented. The results seem to indicate that AC-3_b always performs better than DEE and usually performs better than both AC-3 and AC-7 for the set of testproblems under consideration.

1 Introduction

Arc-consistency algorithms are widely used to reduce the search-space of Constraint Satisfaction Problems (CSPs). In this paper the notion of a *double-support check* is presented to improve the average performance of arc-consistency algorithms. The improvement is that, where possible, consistency-checks are used to find supports for *two* values, one in the domain of each variable, which were previously known to be unsupported. It is motivated by the insight that *in order to minimize the number of consistency-checks it is necessary to maximize the number of uncertainties which are resolved per check.* The idea is used to improve AC-3 and DEE and results in a new general purpose arc-consistency algorithm called AC-3_b. Experimental results of a comparison of AC-3, DEE, AC-3_b and AC-7 are presented. The results seem to indicate that AC-3_b always performs better than DEE and usually performs better than both AC-3 and AC-7 for the set of testproblems under consideration.

The rest of this paper is organised as follows; in Section 2 some concepts used in the Constraint Satisfaction literature are recalled. Related work is discussed in Section 3. Section 4 presents the notion of double-support check. AC-3_b is described in Section 5. Experimental results are presented and discussed in Section 6. Finally, in Section 7, conclusions are presented and further research is discussed.

2 Constraint Satisfaction Theory

This section is an introduction to some basic terminology used in the Constraint Satisfaction literature and the rest of this paper. Section 2.1 is an introduction to some general Constraint Satisfaction terminology. In Section 2.2 the main concepts of arc-consistency are laid out.

2.1 Constraint Satisfaction

CSPs are triples (V, D, C), where V is a set containing the variables of the CSPs, D is a set containing the domains of the variables of the CSPs, and C is a set containing the constraints of the CSPs. The domain of variable k is denoted by D_k. Only binary CSPs are considered, i.e. CSPs where the arity of the constraints is at most 2.

A binary constraint $C_{ij} \subseteq D_i \times D_j$ on variables i and j is a set of pairs. The pairs in the constraint represent the only combinations of values the variables can take. C_{ij} allows for i to take the value v_i and j to take the value v_j iff $(v_i, v_j) \in C_{ij}$. Likewise, a unary constraint is a set of individual values. A member of a constraint is said to *satisfy* the constraint.

A CSP is called *node-consistent* iff, for each variable i, either $C_i \notin C$ or each value in its domain satisfies C_i. Without loss of generality only node-consistent CSPs are considered. Also it is assumed that $(\forall i \in V)(D_i \neq \emptyset)$. A test of the form $v_i \in C_i$ or $(v_i, v_j) \in C_{ij}$ is called a *consistency-check*.

Associated with a binary CSP is its directed constraint graph with nodes corresponding to the variables, and arcs corresponding to the constraints in the CSP. For every unary constraint C_i an arc (i, i) exists. For every binary constraint C_{ij}, two directed arcs (i, j) and (j, i) exist. Two distinct variables i and j in a CSP are called *neighbours* if $C_{ij} \in C$. A CSP is called *connected* if its constraint graph is connected.

An algorithm is called *bi-directional* if it exploits the general property of binary relations that $(v_i, v_j) \in C_{ij} \iff (v_j, v_i) \in C_{ji}$ for any $v_i \in D_i$, any $v_j \in D_j$ and any $C_{ij} \in C$ [1].

2.2 Arc-Consistency

Let i and j be variables, $v_i \in D_i$ and $v_j \in D_j$; then $j = v_j$ *supports* $i = v_i$ if $(v_i, v_j) \in C_{ij}$. In addition $i = v_i$ is said to be *supported* by j if there is some $v_j \in D_j$ s.t. $j = v_j$ supports $i = v_i$.

Given the notion of support, a connected CSP is called *arc-consistent* iff every value in the domain of every variable is supported by all the neighbours of that variable.

A CSP is called *inconsistent* if it has no solutions. Arc-consistency algorithms remove all unsupported values from the domains of variables, or decide that a CSP is inconsistent by finding that some variable has no supported values in its domain.

3 Related Work

This Section describes some related work on *general purpose* arc-consistency algorithms. Here, by general purpose algorithm, is meant an algorithm which can be applied to any CSP.

One of the earliest algorithms is Mackworth's AC-3 [3]. It has a time-complexity of $O(ed^3)$ and a space-complexity of $O(e + nd)$ [4, 5]. As usual n denotes the number of variables in the CSP, e denotes the number of binary-constraints and d denotes the maximum domain-size.

J. Gaschnig describes a related algorithm called DEE in [2]. DEE differs from AC-3 in that, in essence, whereas AC-3 processes only one arc (i, j) at a time, DEE processes both (i, j) and (j, i) at the same time.

A bi-directional arc-consistency algorithm called AC-7 has been presented in [1]. AC-7 has a $O(ed)$ space-complexity, optimal $O(ed^2)$ time-complexity and, if AC-3 and AC-7 are both implemented with the usual lexicographic heuristics, never performs worse than AC-3.

4 Double-Support Checks

This section describes the notion of a *double-support check*. Consider the micro-structure of the 2-variable CSP in Figure 1. If a heuristic of lexicographically ordering the data-structures in AC-3, DEE, and AC-7 is assumed, then AC-7 would need 11 consistency-checks in order to decide that 4 has to be removed from D_A. DEE would also need 11 consistency-checks in order to transform this CSP into its arc-consistent equivalent. For AC-3 this number would be 17. One of the reasons why AC-3 needs far more consistency-checks than DEE and AC-7 is because AC-3 does not exploit the fact that relations are bi-directional. Bi-directionality is used by DEE, because while it is constructing a support for A, each value in D_B which is found to support a value in D_A is marked. It then tries to determine which values in D_B are supported by A but will not try to find a support for those values in D_B which are marked because they are already known to be supported. Bi-directionality is exploited by AC-7 because it never tests for $(b, a) \in C_{ji}$ if it already has checked $(a, b) \in C_{ij}$, and vice versa.

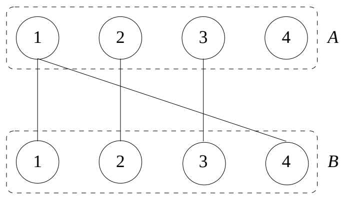

Figure 1: 2-Variable CSP

But even for 2-variable CSPs and the heuristic mentioned above, DEE and AC-7 do too much work. For example, after AC-7 has established that $B = 1$ supports $A = 1$, the first thing it will do in order to find a support for $A = 2$ is to try to find it with $B = 1$. As shown below, it would be better to postpone this, because a support for $A = 2$ may be found elsewhere in D_B, thus allowing for the possibility of two values to be added to those values which are known to be supported, as opposed to only one. The basic idea presented in this paper is the insight that *in order to minimize the number of consistency-checks it is necessary to maximize the number of uncertainties that are resolved per check.*

The objective of arc-consistency processing is to resolve some uncertainty; it has to be known, for each $v_A \in D_A$ and for each $v_B \in D_B$, whether it is supported. Consistency-checks are performed to resolve these uncertainties. A *single-support check*, $(v_A, v_B) \in C_{AB}$, is one in which, before the check is done, it is already known that either v_A or v_B are supported. A *double-support check*, $(v_A, v_B) \in C_{AB}$, is one in which there is still, before the check, uncertainty about the support-status of both v_A and v_B. If a double-support check is successful, two uncertainties are resolved. If a single-support check is successful, only one uncertainty is resolved. A good arc-consistency algorithm, therefore, would always choose to do a double-support check in preference of a single-support check, because the former offers the potential higher payback.

At any stage in the process of making the 2-variable CSP arc-consistent:

- There is a set $S_A^+ \subseteq D_A$ whose values are all known to be supported by B;
- There is a set $S_A^? = D_A \setminus S_A^+$ whose values are unknown, as yet, to be supported by B.

The same holds if the roles for A and B are exchanged. In order to establish support for a value $v_A^? \in S_A^?$ it seems better to try to find a support among the values in $S_B^?$ first, because for each $v_B^? \in S_B^?$ the check $(v_A^?, v_B^?) \in C_{AB}$ is a double-support check and it is just as likely that any $v_B^? \in S_B^?$ supports $v_A^?$ than it is that any $v_B^+ \in S_B^+$ does. Only if no support can be found among the elements in $S_B^?$, should the elements v_B^+ in S_B^+ be used for single-support checks $(v_A^?, v_B^+) \in C_{AB}$. After it has been decided for each value in D_A whether it is supported or not, either $S_A^+ = \emptyset$ and the 2-variable CSP is inconsistent, or $S_A^+ \neq \emptyset$ and the CSP is satisfiable. In the latter case, the elements from D_A which are supported by B are given by S_A^+. The elements in D_B which are supported by A are given by the union of S_B^+ with the set of those elements of $S_B^?$ which further processing will show to be supported by some $v_A^+ \in S_A^+$.

Applying the procedure as sketched above[1] to the example from Figure 1 would lead to a saving of consistency-checks when compared to DEE and AC-7. Instead of the 11 checks needed by AC-7, only the following eight checks would be needed: $(1, 1) \in C_{AB}$, $(2, 2) \in C_{AB}$, $(3, 3) \in C_{AB}$, $(4, 4) \in C_{AB}$, $(4, 1) \in C_{AB}$, $(4, 2) \in C_{AB}$, $(4, 3) \in C_{AB}$, and finally $(4, 1) \in C_{BA}$.

[1] Again a lexicographical ordering is assumed.

```
Q ← {(i,j) ∈ G | i ≠ j};
D' ← copy(D);
while Q ≠ ∅ do begin
    select and remove any (i,j) from Q;
    (S_i^+, S_j^+, S_j^?) ← partition(D'_i, D'_j, C_ij);
    if S_i^+ = ∅ then return (wipeout, ∅);
    if D'_i \ S_i^+ ≠ ∅ then begin
        replace D'_i in D' by S_i^+;
        Q ← Q ∪ {(k,i) ∈ G | k ≠ i, k ≠ j};
    end
    if (j,i) ∈ Q then begin
        remove (j,i) from Q;
        S_j^+ ← S_j^+ ∪ {v_j ∈ S_j^? | ∃v_i ∈ S_i^+ s.t. (v_i, v_j) ∈ C_ij};
        if D'_j \ S_j^+ ≠ ∅ then begin
            replace D'_j in D' by S_j^+;
            Q ← Q ∪ {(k,j) ∈ G | k ≠ i, k ≠ j};
        end
    end
end
return (consistent, D');
```

Figure 2: The AC-3_b Algorithm

It is not difficult to find an example where this approach would lead to more consistency-checks than with AC-7, DEE or AC-3. For a random 2-variable CSP, however, the proposed method is more likely to lead to less consistency-checks. The crucial insight is that *maximizing the number of successful double-support checks is a prerequisite to minimizing the total number of consistency-checks.* The results in Section 6 seem to support this.

5 The AC-3_b Algorithm

Motivated by the observations in Section 4, a new arc-consistency algorithm called AC-3_b is presented. The algorithm is depicted in Figures 2 and 3. The input to the AC-3_b algorithm consists of the directed constraint graph G of the CSP, the set D of the domains of the variables in the CSP and the constraints C of the CSP. Its output is either $(\texttt{wipeout}, \emptyset)$ if the CPS is arc-inconsistent or $(\texttt{consistent}, D')$ otherwise, where D' is the arc-consistent equivalent of D.

As shown in Figure 2 AC-3 uses a function called `partition`. This function is shown in Figure 3. Its input consist of the domains D_i and D_j and the constraint C_{ij}. Its output consists of a tuple $(S_i^+, S_j^+, S_j^?)$ s.t. $S_j^+ \subseteq D_j$, $S_j^? = D_j \setminus S_j^+$ and in addition:

$$
\begin{aligned}
S_i^+ &= \{v_i \in D_i | (\exists v_j \in D_j)((v_i, v_j) \in C_{ij})\} \\
\wedge \quad S_i^+ &= \{v_i \in D_i | (\exists v_j \in S_j^+)((v_i, v_j) \in C_{ij})\} \\
\wedge \quad S_j^+ &\subseteq \{v_j \in D_j | (\exists v_i \in S_i^+)((v_i, v_j) \in C_{ij})\}
\end{aligned}
$$

These rules express the fact that S_i^+ is the set of all values in D_i which are supported by D_j, that each of its members is supported by some value of S_j^+ as well and that S_j^+ does not contain values which are not supported by S_i^+.

```
S_i^? ← D_i;
S_i^+ ← ∅;
S_j^? ← D_j;
S_j^+ ← ∅;
while S_i^? ≠ ∅ do begin
    select and remove any v_i^? from S_i^?;
    if ∃v_j^? ∈ S_j^? s.t. (v_i^?, v_j^?) ∈ C_ij then begin
        select and remove any such v_j^? from S_j^?;
        S_i^+ ← S_i^+ ∪ {v_i^?};
        S_j^+ ← S_j^+ ∪ {v_j^?};
    end
    if ∃v_j^+ ∈ S_j^+ s.t. (v_i^?, v_j^+) ∈ C_ij then S_i^+ ← S_i^+ ∪ {v_i^?};
end
return (S_i^+, S_j^+, S_j^?);
```

Figure 3: The `partition` Algorithm

For the AC-3_b algorithm it is assumed that the input-CSP is already node-consistent. AC-3_b is a refinement of the AC-3 algorithm as described in [3] and DEE as described in [2]. Compared with AC-3 the refinement consists of the fact that if, when arc (i, j) is being processed and the reverse arc (j, i) is also in the queue, then consistency-checks can be saved because only support for the elements in $S_j^?$ has to be found (as opposed to support for all the elements in D_j in the AC-3 algorithm). Compared with DEE the refinement consists of the double-support heuristic.

AC-3_b inherits all its properties like $O(ed^3)$ time-complexity and $O(e+nd)$ space complexity from AC-3. The reader is referred to [7] for proof.

6 Experimental Results

In this Section experimental results are presented to enable comparisons between AC-3, DEE, AC-7 and AC-3_b. In Section 6.1 the experiments and implementation of the algorithms are described. In Section 6.2 the results are discussed. Throughout this section, $\#cc(X)$ denotes the (average) number of consistency-checks performed by algorithm X.

6.1 Description of the Experiment

In order to compare the arc-consistency efficiency of AC-3, DEE, AC-7 and AC-3_b 30,420 random connected CSPs were generated. For each combination of (density,average tightness) in $\{(0.025 * d, 0.025 * t) | t \in \{1, 2, \ldots 39\}, d \in \{1, 2, \ldots 39\}\}$, twenty random connected CSPs were generated. Here, the *density* of a connected binary constraint-network is defined to be $2 \times (e - n + 1)/(n^2 - 3n + 2)$, where n is the number of variables in the CSP and e is the number of edges in the constraint-graph [6]. The *tightness* of a constraint C_{AB} is defined to be $1 - |C_{AB}|/(|D_A| \times |D_B|)$. The *average tightness* of a binary constraint-network is the average of the tightnesses of the binary constraints [6]. The number of variables per CSP was a random number from 15 to 25. The domain size of the variables always equaled the number of variables in the problem. The task of the arc-consistency algorithms consisted of transforming each CSP into its arc-consistent equivalent or deciding that the CSP was arc-inconsistent. The *lexicographical queue* heuristic (see [9] for a description) was used for adding elements to, and removing elements from, queues/streams in all the algorithms.

The DEE version used for the experimentation here, is an arc-queue based version of the one described in [2]. This implementation allows for a good estimation of the efficiency of using double-support checks since DEE and AC-3_b both process the same arcs in the same order. The two algorithms only differ in the way they try to establish a support for the elements in the domains of the variables at both ends of an arc.

6.2 Discussion of Results

The average numbers of consistency-checks for AC-3, DEE, AC-7 and AC-3_b for the random CSPs at each combination of density and tightness are depicted in Figures 4, 5, 6 and 7. The numbers of consistency-checks for each algorithm averaged over each problem are presented in Table 1.

Figures 8, 9, 10, and 11 depict the difference graphs for the average number of consistency-checks between AC-3 and DEE, between AC-3 and AC-3_b, between DEE and AC-3_b, and between AC-7 and AC-3_b. The jagged lines at the bottom of Figures 8, 9, and 11 are where the difference between the number of consistency-checks equals zero. Figure 12 depicts $1 - \#cc(\text{AC-}3_b)/\#cc(\text{DEE})$. Figure 13 depicts $1 - \#cc(\text{AC-}3_b)/\#cc(\text{AC-}7)$.

	DEE	AC-3	AC-3_b	AC-7
$\#cc$	7311	7261	5077	5319

Table 1: Average Number of Consistency-Checks

Table 1 seems to suggest that the DEE approach is a waste. This is because, despite the fact that DEE uses the property that constraints are bi-directional, it can not gain much from it. AC-3, for example, does less work than DEE on some problems because after processing an arc (i, j) AC-3 does not always immediately process the reverse arc (j, i) if it is in the queue, while DEE always does. To postpone processing (j, i) can be good for two reasons. First, if the

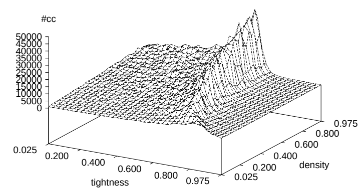

Figure 4: $\#cc$(AC-3)

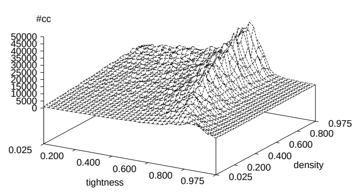

Figure 5: $\#cc$(DEE)

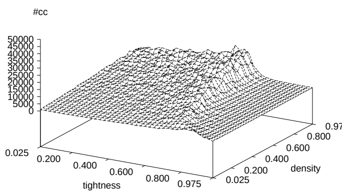

Figure 6: $\#cc$(AC-7)

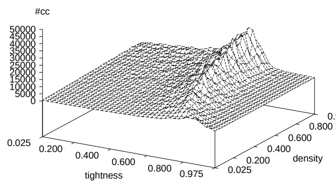

Figure 7: $\#cc$(AC-3_b)

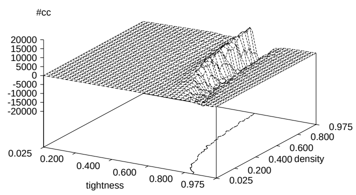

Figure 8: $\#cc$(AC-3) $-$ $\#cc$(DEE)

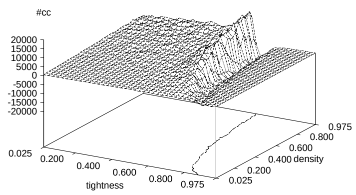

Figure 9: $\#cc$(AC-3) $-$ $\#cc$(AC-3_b)

domain of i gets narrowed several times, the effect of adding the arc (j, i) to the queue several times, can be overcome by processing (j, i) only once. Second, establishing support for j by using values from D_i which will be removed from D_i later may waste consistency-checks. This may be illustrated by the following two possible events. In the first and most extreme case AC-3 would processes another arc, say (i, k), and detects a wipe-out of D_i. Had the arc (j, i) been processed before (i, k) then any consistency-check spent on this arc would have been wasted. In a less extreme case D_i could have been narrowed by processing other arcs to i. This may save work when (j, i) has to be processed because, in general, fewer consistency checks have to be spent on each of the values in D_j when D_i gets smaller. Both effects become more pronounced when constraints become tighter. Only when constraints are loose will DEE outperform AC-3.

AC-3_b is always better than DEE. Figure 10 shows this–by the way, note that, those parts of the surface of the graph in Figure 10 which appear to be in the horizontal plane $\#cc = 0$ are, in fact, above this plane. This seems to suggest that the strategy of using double-support

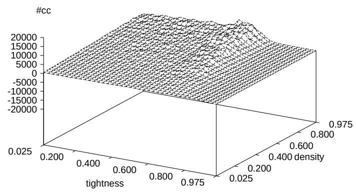

Figure 10: $\#cc$(DEE) $-$ $\#cc$(AC-3_b)

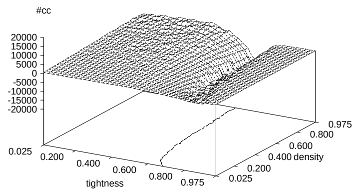

Figure 11: $\#cc$(AC-7) $-$ $\#cc$(AC-3_b)

checks to establish supports is a good one. It is interesting to see that the ratio between the number of consistency-checks saved by the double-support heuristic and the total number of consistency-checks, is nearly constant for fixed tightness (see Figure 12).

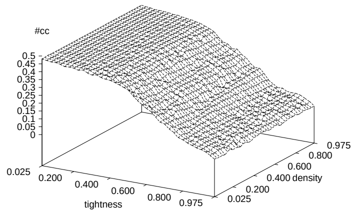

Figure 12: $1 - \frac{\#cc(\text{AC-3}_b)}{\#cc(\text{DEE})}$

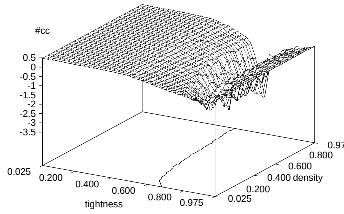

Figure 13: $1 - \frac{\#cc(\text{AC-3}_b)}{\#cc(\text{AC-7})}$

AC-3_b is better than AC-3 everywhere to the left of the phase transition region. AC-3 becomes better in the phase transition region, and stays better from there onwards. The reason for this is the same as why AC-3 becomes better than DEE as average tightness increases.

Nevertheless, it seems that AC-3_b is a more efficient algorithm than AC-3. The disadvantage of always processing two arcs (the “DEE effect”) is turned into an advantage by adopting the double-support heuristic. Possibilities seem to exist to improve AC-3_b and DEE. One possibility is to force the algorithms to degenerate to AC-3 (i.e. never to process a double arc) as soon as they know (or learn) they are processing tight constraints.

AC-3_b requires less consistency-checks than AC-7 in a larger area in the problem space (see Figures 11 and 13) but as tightness increases AC-7 becomes better. In the low tightness area AC-3_b does better than AC-7 because most of its consistency-checks will lead to a double support. AC-7 accumulates knowledge about consistency-checks it has already carried out and never repeats one. Therefore it has to outperform AC-3_b at some stage as tightness increases.

It may seem surprising that AC-3_b seems to perform better on the test problems than AC-7, despite the fact that AC-3_b has a worse time-complexity. However, this phenomenon actually also occurs elsewhere. For example, in the linear programming community the exponential simplex algorithm is still preferred over existing polynomial algorithms because it behaves better on average. AC-4 (another arc-consistency algorithm) has a better time complexity than AC-3. This did not stop people from using AC-3—It was almost always better than AC-4 [8].

7 Conclusions and Recommendations

The notion of a double-support check heuristic has been presented. Experiments seem to suggest that this heuristic can be used to improve the average performance of AC-3. The resulting improved arc-consistency algorithm called AC-3_b has been presented. Experimental results seem to indicate that for the problems under consideration AC-3_b is more efficient in a large part of the tightness-density space than any existing arc-consistency algorithm. Possibilities exist to improve AC-3_b in the high tightness area.

It seems that double-supports can be used to improve AC-7 as well. One of the changes to the algorithm should consist adding a dynamic value ordering for the values in the domains of the variables. This ordering should partially depend on consistency-checks which were previously carried out, and should also consist of a tie-break ordering. Future research will have to learn what these proposed changes to these algorithms will mean in terms of average performance.

8 Acknowledgements

The work reported here was funded by the European Commission under ESPRIT project number 20501, with acronym CEDAS.

References

[1] C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *IJCAI'95*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.

[2] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the Second Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.

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

[4] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–73, 1985.

[5] A.K. Mackworth and E.C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59:57–62, 1993.

[6] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *ECAI'94*, pages 125–129. John Wiley & Sons, 1994.

[7] M.R.C. van Dongen. AC-3_b, an efficient arc-consistency algorithm with low space-complexity. Technical Report TR-97-01, Department of Computer Science, National University of Ireland, Cork, College Road, Cork, Ireland, 1997.

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

[9] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI '92*, pages 163–169, Vancouver, British Columbia, Canada, 1992.