AC-3_d an Efficient Arc-Consistency Algorithm with a Low Space-Complexity

M.R.C. van Dongen
dongen@cs.ucc.ie

Cork Constraint Computation Centre[1]
CS Department UCC
Western Road
Cork
Ireland

Technical Report TR-01-2002

June 2002

Available from http://csweb.ucc.ie/~dongen/papers/4C/02/4C-01-2002.pdf

[1]The Cork Constraint Computation Centre is supported by Science Foundation Ireland

Abstract

Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). They use *support-checks* to find out about the properties of CSPs. They use *arc-heuristics* to select the constraint and *domain-heuristics* to select the values for their next support-check. We will demonstrate that domain-heuristics can significantly enhance the average time-complexity of existing arc-consistency algorithms. We will combine Alan Mackworth's AC-3 and John Gaschnig's DEE and equip the resulting hybrid with a *double-support* domain-heuristic thereby creating an arc-consistency algorithm called AC-3_d, which has an average time-complexity which can compete with AC-7 and which improves on AC-7's space-complexity. AC-3_d is easy to implement and requires the same data structures as AC-3. We will present experimental results to justify our average time-complexity claim.

Chapter 1

Introduction

Arc-consistency algorithms are widely used to prune the search-space of Constraint Satisfaction Problems (CSPs). We will demonstrate that *domain-heuristics* can significantly enhance the average time-complexity of existing arc-consistency algorithms. We will integrate Alan Mackworth's AC-3 and John Gaschnig's DEE and equip the resulting hybrid with a so-called *double-support* domain-heuristic thereby creating a general purpose arc-consistency algorithm called AC-3_d, which has an average time-complexity which can compete with AC-7—the current state-of-the-art in arc-consistency algorithms—and improves on AC-7's space-complexity. We will present theoretical and experimental results to justify our average time-complexity claim.

One reason for the increased performance of AC-3_d is that it uses the double-support heuristic as opposed to the so-called *lexicographical* domain-heuristic, which is the most commonly used domain-heuristic for arc-consistency algorithms. To find out about the difference between these two heuristics we will study two algorithms called $\mathcal{L}$ and $\mathcal{D}$. $\mathcal{L}$ uses the lexicographical heuristic and $\mathcal{D}$ uses the double-support heuristic. We will assume that there are only two variables in the CSP. As a and b become large $\mathcal{L}$ will require about $2a + 2b - 2\log_2(a) - 0.665492$ checks on average. Here a and b are the domain-sizes and $\log_2(\cdot)$ is the base-2 logarithm. If $a + b \geq 14$ then $\mathcal{D}$ will require an average number of support-checks which is strictly below $2\max(a, b) + 2$. We will also provide an "optimality" result which indicates that if $a + b \geq 14$ then no algorithm can save two checks or more on average than $\mathcal{D}$. Our results indicate that as a domain-heuristic the double-support heuristic is to be preferred to the lexicographical heuristic, which is the most commonly used domain-heuristic for arc-consistency algorithms. Our optimality result is informative about the possibilities and limitations of domain-heuristics for arc-consistency algorithms.

We will present experimental results from a comparison between AC-3_d which has a $\mathbf{O}(e + nd)$ space-complexity and AC-7. As usual n is the number of variables, e is the number of constraints, and d is the maximum domain size. AC-7 has a $\mathbf{O}(ed)$ space-complexity and has proved to require fewer support-checks than other general purpose arc-consistency algorithms. Our comparison involves random problem and problems from the constraint literature. The results of the comparison demonstrate that, for the problems under consideration, AC-3_d can compete with AC-7 both in time on the wall and in the number of support-checks.

The remainder of this paper is organised as follows. In Chapter 2 we shall provide basic

definitions and review constraint satisfaction. The lexicographical and double-support heuristics will be presented in Chapter 3. In Chapter 4 we shall present our average time-complexity results. In Chapter 5 we shall present AC-3_d. We shall present our experimental results in Chapter 6. Our conclusions will be presented in Chapter 7.

Chapter 2

Constraint Satisfaction

2.1 Basic Definitions

A *Constraint Satisfaction Problem* (or CSP) is a tuple (X, D, C), where X is a set containing the variables of the CSP, D is a function which maps each of the variables in X to its domain, and C is a set containing the constraints of the CSP.

Let (X, D, C) be a CSP and let $\alpha \in X$ and $\beta \in X \setminus \{\alpha\}$ be two variables. Furthermore, let $D(\alpha) = \{1, \ldots, a\} \neq \emptyset$ be the domain of α, and let $D(\beta) = \{1, \ldots, b\} \neq \emptyset$ be the domain of β. A (binary) constraint $M \in C$ between α and β is an a by b zero-one matrix, i.e. it is a matrix with a rows and b columns whose entries are either zero or one. A tuple (i, j) in the Cartesian product of the domains of α and β is said to *satisfy* the constraint M if $M_{ij} = 1$. Here M_{ij} is the j-th column of the i-th row of M. A value $i \in D(\alpha)$ is said to be *supported* by $j \in D(\beta)$ if $M_{ij} = 1$. Similarly, $j \in D(\beta)$ is said to be supported by $i \in D(\alpha)$ if $M_{ij} = 1$. M is called *arc-consistent* if for every $i \in D(\alpha)$ there is a $j \in D(\beta)$ which supports i and vice versa. A CSP is called arc-consistent if the domains of its variables are non-empty and its constraints are arc-consistent. A variable α is a *neighbour* of variable β if there is a binary constraint between α and β. The *degree* of α is denoted $\deg(\alpha)$. It is defined as the number of neighbours of α.

The *density* of a constraint-graph with $n > 1$ nodes and e edges is sometimes defined as $2e/(n^2 - n)$. For the duration of this paper and for compatibility reasons we shall stick to this definition. The *tightness* of an a by b constraint M is defined as $1 - \frac{1}{ab} \sum_{i=1}^{a} \sum_{j=1}^{b} M_{ij}$.

We will denote the set containing all a by b zero-one matrices by $\mathbb{M}^{ab}$. We will call matrices, rows of matrices, and columns of matrices *non-zero* if they contain more than zero ones, and will call them *zero* otherwise.

The *row-support* (*column-support*) of a matrix is the set containing the indices of its non-zero rows (columns). An *arc-consistency algorithm* removes all the unsupported values from the domains of the variables of a CSP until this is no longer possible. A *support-check* is a test to find the value of an entry of a matrix. We will write $M^{?}_{ij}$ for the support-check to find the value of M_{ij}. An arc-consistency algorithm has to carry out the support-check $M^{?}_{ij}$ to find out about the value of M_{ij}. The time-complexity of arc-consistency algorithms is expressed in the number of support-checks they require to find the supports of their arguments.

Let $\mathcal{A}$ be an arc-consistency algorithm and let M be an a by b matrix. We will write $\text{checks}_{\mathcal{A}}(M)$ for the number of support-checks required by $\mathcal{A}$ to compute the row and column-support of M.

The *total number of support-checks* of $\mathcal{A}$ over M^{ab} is the function $\text{total}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$ defined as $\text{total}_{\mathcal{A}}(a,b) = \sum_{M \in \mathbb{M}^{ab}} \text{checks}_{\mathcal{A}}(M)$. The *average time-complexity* of $\mathcal{A}$ *over* $\mathbb{M}^{ab}$ is the function $\text{avg}_{\mathcal{A}} : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Q}$ defined as $\text{avg}_{\mathcal{A}}(a,b) = \text{total}_{\mathcal{A}}(a,b)/2^{ab}$.

A support-check $M_{ij}^{?}$ is said to *succeed* if $M_{ij} = 1$ and said to *fail* otherwise. If a support-check succeeds it is called *successful* and *unsuccessful* otherwise. Note that it follows from our definition of average time-complexity that it is just as likely for a support-check to succeed as it is for it to fail.

$M_{ij}^{?}$ is called a *single-support check* if, just before the check was carried out, the row-support status of i was known and the column-support status of j was unknown, or vice versa. A successful single-support check $M_{ij}^{?}$ leads to new knowledge about one thing. Either it leads to the knowledge that i is in the row-support of M where this was not known before the check was carried out, or it leads to the knowledge that j is in the column-support of M where this was not known before the check was carried out. $M_{ij}^{?}$ is called a *double-support check* if, just before the check was carried out, both the row-support status of of i and the column-support status of j were unknown. A successful double-support check $M_{ij}^{?}$ leads to new knowledge about *two* things. It leads to the knowledge that i is in the row-support of M and that j is in the column-support of M where neither of these facts were known to be true just before the check was carried out. A domain-heuristic is called a *double-support heuristic* if it prefers double-support checks to other checks.

On average it is just as likely that a random double-support check will succeed as it is that a random single-support check will succeed—in both cases one out of two checks will succeed on average. However, the potential payoff of a double-support check is twice as large that of a single-support check. This is an indication that at domain-level arc-consistency algorithms should prefer double-support checks to single-support checks.

Another indication that arc-consistency algorithms should prefer double-support checks is that in order to minimise the total number of support-checks it is a necessary condition to maximise the number of successful double-support checks [van Dongen, 2002].

2.2 Related Literature

In 1977 Mackworth presented an arc-consistency algorithm called AC-3 [Mackworth, 1977]. Together with Freuder he presented a lower bound of $\mathbf{\Omega}(ed^2)$ and an upper bound of $\mathbf{O}(ed^3)$ for its worst-case time-complexity [Mackworth and Freuder, 1985]. The algorithm, as mentioned before, has a $\mathbf{O}(e + nd)$ space-complexity.

AC-3, as presented by Mackworth, is not an algorithm as such; it is a *class* of algorithms which have certain data-structures in common and treat them similarly. The most prominent data-structure used by AC-3 is a *queue* which initially contains each of the pairs (α, β) and (β, α) for which there exists a constraint between α and β. The basic machinery of AC-3 is

such that *any* tuple can be removed from the queue. For a "real" implementation this means that heuristics determine the choice of the tuple that is going to be removed from the queue. By selecting a member from the queue, these heuristics determine the constraint that will be used for the next support-checks. Such heuristics will be called *arc-heuristics*.

Not only are there arc-heuristics for AC-3, but also are there heuristics which, given a constraint, select the values in the domains of the variables that will be used for the next support-check. Such heuristics we will call *domain-heuristics*.

Experimental results from Wallace and Freuder clearly indicate that arc-heuristics influence the average performance of arc-consistency algorithms [Wallace and Freuder, 1992]. Gent *et al* have made similar observations [Gent *et al.*, 1997].

Bessière, Freuder and Régin present another class of arc-consistency algorithms called AC-7 [Bessière *et al.*, 1995; Bessière *et al.*, 1999]. AC-7 is an instance of the AC-INFERENCE schema. AC-7 saves support-checks by making inference. Inference is made at domain-level, where it is exploited that $M_{ij} = M_{ji}^T$. Here $\cdot^T$ denotes transposition. AC-7 has an optimal upper bound of $\mathbf{O}(ed^2)$ for its worst-case time-complexity and has been reported to behave well on average. AC-7's space-complexity is $\mathbf{O}(ed)$.

In their papers, Bessière, Freuder and Régin present experimental results that the AC-7 approach is superior to the AC-3 approach if the number of support-checks is concerned. If the time on the wall is concerned, they observe that AC-3 is a good alternative if checks are cheap [Bessière *et al.*, 1999]. They present results of applications of MAC-3 and MAC-7 to real-world problems. Here MAC-i is a backtracking algorithm which uses AC-i to maintain arc-consistency during search [Sabin and Freuder, 1994].

Van Dongen and Bowen present results from an experimental comparison between AC-7 and AC-3_b, which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEE [van Dongen and Bowen, 2000; Mackworth, 1977; Gaschnig, 1978]. Their comparison only considered support-checks. In their setting AC-3_b was equipped with a lexicographical arc-heuristic. At the domain-level AC-3_b uses a double-support heuristic. In their setting, AC-7 was equipped with a lexicographical arc-heuristic and domain-heuristic. AC-3_b has the same worst-case time-complexity as AC-3. In van Dongen and Bowen's setting it turned out that AC-3_b was more efficient than AC-7 for the majority of their $30,420$ random problems. Also AC-3_b was more efficient on average. These are surprising results because AC-3_b, unlike AC-7, has to repeat support-checks because it cannot remember them. They are also interesting because AC-3_b has a better space-complexity than AC-7 ($\mathbf{O}(e + nd)$ versus $\mathbf{O}(ed)$). These results were the first indication that domain-heuristics can improve arc-consistency algorithms.

Chapter 3

Two Arc-Consistency Algorithms

In this section we shall introduce the two arc-consistency algorithms $\mathcal{L}$ and $\mathcal{D}$. The two algorithms only differ in their domain-heuristic. $\mathcal{L}$ uses a *lexicographical heuristic* and $\mathcal{D}$ uses a *double-support heuristic*. We shall first present $\mathcal{L}$ and then present $\mathcal{D}$.

From here on we shall sometimes use $\mathcal{L}$ ($\mathcal{D}$) instead of using lexicographical domain-heuristic (double-support domain-heuristic) and vice versa. This is justified because the algorithm completely determines the heuristic and vice versa.

3.1 The Lexicographical Algorithm $\mathcal{L}$

$\mathcal{L}$ does not repeat support-checks. It first tries to establish its row-support. It does this for each row in the lexicographical order on the rows. When it seeks support for row r it tries to find the lexicographically smallest column which supports r. After $\mathcal{L}$ has computed its row-support, it tries to find support for those columns whose support-status is not yet known. It does this in the lexicographical order on the columns. When $\mathcal{L}$ tries to find support for a column c, it tries to find it with the lexicographically smallest row that was not yet known to support c.

Pseudo-code for $\mathcal{L}$ is depicted in Figure 3.1. $\mathcal{L}$ computes its support by removing the unsupported rows and columns from the domains of the variables using two procedures $\mathrm{remove\_row}$ and $\mathrm{remove\_col}$. Without loss of generality we have assumed that the size of the constraints is determined by the domains.

$\mathcal{L}$ is, in essence, Gaschnig's version of Mackworth's revise algorithm which is part of the AC-3 machinery [Gaschnig, 1978; Mackworth, 1977]. $\mathcal{L}$ only establishes row-support whereas revise establishes both row-support and column-support.

$\mathcal{L}$ has been presented so as to highlight its essence. Using standard techniques the algorithm can be transformed to use counters instead of the two-dimensional array $\mathrm{checked}$. With the use of this technique the algorithm will have a $\mathbf{O}\,(a+b)$, that is, a $\mathbf{O}\,(d)$ space-complexity [Bessière *et al.*, 1995]. It is a straightforward exercise to prove correctness of $\mathcal{L}$.

```
constant UNSUPPORTED = −1;

procedure L( a by b constraint M ) = begin
    /* Initialisation. */
    for each row r do begin
        rsupp[r] ≔ UNSUPPORTED;
        for each column c do
            checked[r][c] ≔ UNSUPPORTED;
    end;
    for each column c do
        csupp[c] ≔ UNSUPPORTED;

    /* Find row-support. */
    for each row r do begin
        c ≔ 1;
        while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
            if (M?rc = 1) then begin
                rsupp[r] ≔ c;
                csupp[c] ≔ r;
            end;
            c ≔ c + 1;
        end;
        if (rsupp[r] = UNSUPPORTED) then
            remove_row(r);
    end;

    /* Complete column-support. */
    for each column c do begin
        r ≔ 1;
        while (r ≤ a) and (csupp[c] = UNSUPPORTED) do begin
            if (not checked[r][c]) then
                if (M?rc = 1) then
                    csupp[c] ≔ r;
            r ≔ r + 1;
        end;
        if (csupp[c] = UNSUPPORTED) then
            remove_col(c);
    end;
end;
```

Figure 3.1: Algorithm $\mathcal{L}$

```
constant UNSUPPORTED = −1;
constant SINGLE = −2;
constant DOUBLE = −3;

procedure 𝒟( a by b constraint M ) = begin
    /* Initialisation. */
    for each row r do begin
        rkind[r] ≔ UNSUPPORTED;
        rsupp[r] ≔ UNSUPPORTED;
    end;
    for each column c do
        csupp[c] ≔ UNSUPPORTED;

    /* Find row-support. */
    for each row r do begin
        c ≔ 1;
        /* First try to find support for r using double-support checks. */
        while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
            if (csupp[c] = UNSUPPORTED) then begin
                /* M?rc is a double-support check. */
                if (M?rc = 1) then begin
                    rkind[r] ≔ DOUBLE;
                    rsupp[r] ≔ c;
                    csupp[c] ≔ r;
                end
                c ≔ c + 1;
            end;
        c ≔ 1;
        /* If r is still unsupported then try to find support using single-support checks. */
        while (c ≤ b) and (rsupp[r] = UNSUPPORTED) do begin
            if (csupp[c] ≠ UNSUPPORTED) then begin
                /* M?rc is a single-support check. */
                if (M?rc = 1) then begin
                    rkind[r] ≔ SINGLE;
                    rsupp[r] ≔ c;
                end;
            end;
            c ≔ c + 1;
        end;
        if (rsupp[r] = UNSUPPORTED) then
            remove_row(r);
    end;

    /* Complete column-support. */
    for each column c do begin
        r ≔ 1;
        while (r ≤ a) and (csupp[c] = UNSUPPORTED) do begin
            if (rsupp[r] < c) and (rkind[r] = DOUBLE) then
                if (M?rc = 1) then begin
                    csupp[c] ≔ r;
                    rsupp[r] ≔ c;
                end;
            r ≔ r + 1;
        end
        if (csupp[c] = UNSUPPORTED) then
            remove_col(c);
    end;
end;
```

Figure 3.2: Algorithm $\mathcal{D}$

3.2 The Double-Support Algorithm $\mathcal{D}$

In this section we shall introduce the second arc-consistency algorithm called $\mathcal{D}$. $\mathcal{D}$ uses a double-support heuristic as its domain-heuristic. The heuristic has underlying lexicographical heuristics to break ties.

$\mathcal{D}$'s strategy is a bit more complicated than that of $\mathcal{L}$. Like $\mathcal{L}$ it does not repeat support-checks. The algorithm will first find its row-support in the lexicographical order on its rows. When it tries to find support for row r it will first use double-support checks. It does this by finding the lexicographically smallest column c whose support-status is not yet known. When there are no more double-support checks left then $\mathcal{D}$ will use single-support checks to find support for row r. Finally, $\mathcal{D}$ will use single-support checks for the columns for which no support had been established.

We have depicted pseudo-code for $\mathcal{D}$ in Figure 3.2. $\mathcal{D}$ is implemented in a $\mathbf{O}\,(a+b)$, that is a $\mathbf{O}\,(d)$, space-complexity.

It easy to prove that $\mathcal{D}$ does compute its row-support correctly. To prove that it also computes its column-support correctly is not much more difficult. Immediately after $\mathcal{D}$ has established its row-support, for every row r each of the following holds:

1. $\mathrm{rkind}[r] \neq \mathrm{DOUBLE}$ and $\mathrm{rsupp}[r] = \mathrm{UNSUPPORTED}$ if and only if for every column c the check $M^?_{rc}$ has been carried out and failed;

2. $\mathrm{rkind}[r] \neq \mathrm{DOUBLE}$ and $\mathrm{rsupp}[r] \neq \mathrm{UNSUPPORTED}$ if and only if r's support was established with a single-support check. Note that for every unsupported column c the check $M^?_{rc}$ has already been carried out while $\mathcal{D}$ tried to establish support for r using double-support checks;

3. $\mathrm{rkind}[r] = \mathrm{DOUBLE}$ if and only if r's support was established with a double-support check. Note that for every unsupported column $c \leq \mathrm{rsupp}[r]$ the check $M^?_{rc}$ has already been carried out and that for every unsupported column $c > \mathrm{rsupp}[r]$ the check $M^?_{rc}$ has not been carried out.

Together, Point 1 and 2 imply that if $\mathrm{rkind}[r] \neq \mathrm{DOUBLE}$ then for each unsupported column c the check $M^?_{rc}$ has already been carried out. To complete its column support $\mathcal{D}$ therefore only considers rows r for which $\mathrm{rkind}[r] = \mathrm{DOUBLE}$ and unsupported columns c for which $\mathrm{rsupp}[r] < c$. The remainder of the proof is easy because Point 3 is maintained as an invariant while $\mathcal{D}$ completes the computation of its unsupported columns.

It is important to point out that the first two for-each statements in the $\mathcal{D}$ algorithm can be avoided. The statements in the first for-each statement can be integrated with the third for-each statement by putting them at the start of that third for-each statement statement. The statements can be put just before ore just after the assignment $c \coloneqq 1$. The second for-each statement can be avoided by representing a value c in the domains of a variable by a tuple $(\,c, \mathrm{csupp}[c]\,)$. The second members of the tuples should be initialised to $\mathrm{UNSUPPORTED}$ before the first call to $\mathcal{D}$. A good time to do this is when the domains are initialised. Furthermore, the second members of the tuples should be set it to $\mathrm{UNSUPPORTED}$ at the end of the last for-each statement. A

good way to do this is by adding an extra else-clause for the last if-statement (it corresponds to the case where $\mathrm{csupp}[c] \neq \mathrm{UNSUPPORTED}$). This will ensure that each second member of the tuples is $\mathrm{UNSUPPORTED}$ at the start of every call to $\mathcal{D}$. The last transformation does not affect the space-complexity.

Chapter 4

Average Time-Complexity Results

In this section we shall present average time-complexity results for $\mathcal{L}$ and $\mathcal{D}$. The proofs are long and tedious and due to space limitations they have been omitted. The reader is referred to [van Dongen, 2002] for proof and further information.

The remainder of this section is as follows. We shall first present an *exact* formula and a good approximation for $\mathrm{avg}_{\mathcal{L}}(a,b)$. Next we shall present an *exact* formula and a tight upper bound for $\mathrm{avg}_{\mathcal{D}}(a,b)$. Finally, we shall present an "optimality" result for $\mathrm{avg}_{\mathcal{D}}(a,b)$ and compare $\mathrm{avg}_{\mathcal{L}}(a,b)$ and $\mathrm{avg}_{\mathcal{D}}(a,b)$.

Theorem 4.1 (Average Time Complexity of $\mathcal{L}$) *Let a and b be positive integers. The average time-complexity $\mathrm{avg}_{\mathcal{L}}(a,b)$ of $\mathcal{L}$ over $\mathbb{M}^{ab}$ is given by*

$$\mathrm{avg}_{\mathcal{L}}(a,b) = a(2-2^{1-b}) + (1-b)2^{1-a} + 2\sum_{c=2}^{b}(1-2^{-c})^a.$$

Following [Flajolet and Sedgewick, 1996, Page 59] we obtain the following accurate estimate:

$$\mathrm{avg}_{\mathcal{L}}(a,b) \approx \widetilde{\mathrm{avg}}_{\mathcal{L}}(a,b) = 2a + 2b - 2\log_2(a) - 0.665492.$$

Here, $\log_2(\cdot)$ is the base-2 logarithm. This estimate is already good for relatively small a and b. For example, for $a = b = 10$ we have $|\mathrm{avg}_{\mathcal{L}}(a,b) - \widetilde{\mathrm{avg}}_{\mathcal{L}}(a,b)|/\mathrm{avg}_{\mathcal{L}}(a,b) < 0.5\%$.

Theorem 4.2 (Average Time Complexity of $\mathcal{D}$) *Let a and b be non-negative integers. The average time-complexity $\mathrm{avg}_{\mathcal{D}}(a,b)$ of $\mathcal{D}$ over $\mathbb{M}^{ab}$ is given by $\mathrm{avg}_{\mathcal{D}}(a,b)$ if $a = 0$ or $b = 0$, and by*

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a,b) &= 2 + (b-2)2^{1-a} + (a-2)2^{1-b} + 2^{2-a-b} - (a-1)2^{1-2b} \\ &\quad + 2^{-b}\,\mathrm{avg}_{\mathcal{D}}(a-1,b) + (1-2^{-b})\,\mathrm{avg}_{\mathcal{D}}(a-1,b-1)\end{aligned}$$

if $a \neq 0$ and $b \neq 0$.

Let a and b be positive integers such that $a + b \geq 14$. The following upper bound for $\mathrm{avg}_{\mathcal{D}}(a, b)$ is presented in [van Dongen, 2001; 2002]:

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a,b) \;<\; & 2\max(a,b) + 2 \\ & - (2\max(a,b) + \min(a,b))2^{-\min(a,b)} \\ & - (2\min(a,b) + 3\max(a,b))2^{-\max(a,b)}.\end{aligned}$$

It is a relatively easy exercise to prove that a value that requires support from a domain of size d requires $2^{-d}\sum_{i=1}^{d} i2^{d-i} = 2 - 2^{1-d} \approx 2$ checks on average [van Dongen, 2002]. As a consequence *any* arc-consistency algorithm will require at least about $2\max(a, b)$ checks on average. We can use this and the upper bound for $\mathrm{avg}_{\mathcal{D}}(a, b)$ to derive the important result that $\mathcal{D}$ is "almost optimal," because if $14 \leq a + b$, then $\mathrm{avg}_{\mathcal{D}}(a, b) - \mathrm{avg}_{\mathcal{A}}(a, b) < 2$ for any arc-consistency algorithm $\mathcal{A}$.

It is not difficult to see that the *minimum* number of support-checks required by $\mathcal{L}$ is $a + b - 1$. This implies that if $a + b \geq 14$ and if a and b are approximately the same then the *minimum* number of support-checks required by $\mathcal{L}$ is almost the same as the *average* number of support-checks required by $\mathcal{D}$!

$\mathcal{D}$ is a better algorithm than $\mathcal{L}$ because its upper bound is lower than the bound that we derived for $\mathcal{L}$. When a and b get large and are of the same magnitude then the difference is about $a + b - 2\log_2((a + b)/2)$ which is quite substantial.

At this point it may be interesting to state that for each algorithm $\mathcal{A} \in \{\mathcal{L}, \mathcal{D}\}$ we have verified the formula for $\mathrm{avg}_{\mathcal{A}}(a, b)$ for $1 \leq a, b \leq 6$. We did this by applying $\mathcal{A}$ to each of the matrices in $\mathbb{M}^{ab}$ and by computing the total number $T_{\mathcal{A}}$ of support-checks which were required. We have verified that $T_{\mathcal{A}}$ was exactly $2^{ab}\,\mathrm{avg}_{\mathcal{A}}(a, b)$. This is comforting because $\mathrm{avg}_{\mathcal{A}}(a, b)$ is defined as $\sum_{M\in\mathbb{M}^{ab}} \mathrm{checks}_{\mathcal{A}}(M)/2^{ab}$ and if our analysis is correct then this ought to be exactly $T_{\mathcal{A}}/2^{ab}$.

Chapter 5

The AC-3$_d$ Algorithm

In this section we shall briefly describe AC-3$_d$ and sketch a correctness proof. We assume familiarity of the reader with AC-3.

```
Q := set containing all arcs in the constraint graph;

while (Q ≠ ∅) do begin
   select and remove any arc ( α, β ) from Q;
   if ( β, α ) is also in Q then begin
      remove ( β, α ) from Q;
      use 𝒟 to simultaneously revise D(α) and D(β) using the constraint between α and β;
      if (D(α) = ∅) then
         return wipeout;
      else begin
         if D(α) has changed then
            for each neighbour γ ≠ β of α do Q := Q ∪ { ( γ, α ) };
         if D(β) has changed then
            for each neighbour γ ≠ α of β do Q := Q ∪ { ( γ, β ) };
      end
   end
   else begin
      use Mackworth's revise to revise D(α) using the constraint between α and β;
      if (D(α) = ∅) then
         return wipeout;
      else if D(α) has changed then
         for each neighbour γ ≠ α of β do Q := Q ∪ { ( γ, β ) };
   end
end;
```

Figure 5.1: Algorithm AC-3$_d$

We have depicted AC-3$_d$ in Figure 5.1. The machinery of AC-3$_d$ is inspired by Mackworth's AC-3 and Gashnig's DEE [Mackworth, 1977; Gaschnig, 1978]. AC-3$_d$ uses a queue of arcs just like AC-3. If AC-3$_d$'s arc-heuristics select the arc (α, β) from the queue and if the reverse arc (β, α) is not in the queue then AC-3$_d$ proceeds like AC-3 by *revising* the domain $D(\alpha)$ of α using the constraint M between α and β. Here, to revise a domain using constraint M, means to remove the unsupported values from that domain using the constraint M. AC-3$_d$ uses Mackworth's revise to revise $D(\alpha)$ with M. If the domain $D(\alpha)$ of α has changed due to the

revision then for each neighbour $\gamma \neq \beta$ of α the arc (γ, α) is added to the queue if it was not in the queue. The difference between AC-3 and AC-3_d becomes apparent when AC-3_d's arc-heuristic selects the arc (α, β) from the queue and when the reverse arc (β, α) is also in the queue. If this is this case then AC-3_d also removes (β, α) from the queue and uses $\mathcal{D}$ to simultaneously revise the domains of α and β. Arcs are added to the queue in a similar way as described before.

In the implementation of AC-3_d it is required to find out if the domain of a variable has changed as a result of a revision. Mackworth's revise already allows for this. For $\mathcal{D}$ this is not, yet, possible. However, if we add two additional arguments to $\mathcal{D}$, one for each of the domains, which are set to FALSE by $\mathcal{D}$ if no change occurred to the domain they belong to and to TRUE otherwise then we can cheaply find out about changes to these domains. The additional overhead to $\mathcal{D}$ does not change its time-complexity.

AC-3_d inherits its $\mathbf{O}(ed^3)$ worst-case time-complexity and $\mathbf{O}(e + nd)$ space-complexity from AC-3_b and from the fact that $\mathcal{D}$ has a $\mathbf{O}(d)$ space-complexity.

Chapter 6

Experimental Results

In this section we present some results from a comparison of AC-3_d against AC-3 and AC-7. The organisation of this section is as follows. In Section 6.1 we observe that there is a problem in the literature which concerns the reproducibility of experiments which involve MAC solvers. In Section 6.2 we shall describe the experiment. In Section 6.3 we shall present and discuss our results A summary will be presented in Section 6.4.

6.1 Reproducability

One of the most important applications of arc-consistency is MAC search. Therefore, arc-consistency algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ are frequently compared by: embedding $\mathcal{A}_i$ into MAC-algorithm $\mathcal{M}_i$, for $i \in \{ 1, 2 \}$, and to use the results of the comparison between $\mathcal{M}_1$ and $\mathcal{M}_2$ to compare $\mathcal{A}_1$ and $\mathcal{A}_2$. To compare the ratio between the support-checks that were required by $\mathcal{A}_1$ and $\mathcal{A}_2$ one simply divides the checks that were required by $\mathcal{M}_1$ by the checks that were required by $\mathcal{M}_2$. To compare time on the wall one uses a similar approach.

This does not always result in a fair comparison. For example, $\mathcal{M}_1$ and $\mathcal{M}_2$ may use a different variable ordering heuristic during search. Even with commonly used heuristics there may still be differences. For example, a standard minimum domain size ordering heuristic which uses a maximum degree ordering as a tie-breaker does not rule out the possibility of ties and the choice of the next variable can make the difference between finding an easy solution or getting lost in the search space. For RLFPA#11, for example, for *a* minimum domain size variable ordering heuristic our MAC-3 solver required 1,453 seconds, 1,191,650,012 support-checks, and 1,928,872 backtracks to find the first solution on a 1000 MHz DELL Latitude,[1] whereas Bessière, Freuder and Régin report a solution time of 36.31 seconds for their MAC-3 solver on a 200 MHz PC [Bessière *et al.*, 1999]. It is obvious that the difference between the two is not a representative of the difference between the underlying arc-consistency components.

Gomes, Selman, and Crato study the effects of some commonly used variable ordering heuristics on the variability in time to find the first solution in backtrack search. [Gomes *et al.*, 1997]. They observe that anomalies do occur and present remedies to overcome them.

[1] Our MAC-3_d solver required 990 seconds, 582,777,218 support-checks, and 1,928,872 backtracks.

Exact information about variable ordering heuristics for MAC search is very scarce in the literature. This is a serious problem. We hope that future papers will contain sufficient information about these heuristics so as to facilitate easy reproducibility.

6.2 The Experiment

We do not have our own implementation of AC-7. It is for this reason and for reasons as laid out in Section 6.1 that we decided not to use MAC searchers to compare their underlying arc-consistency components. To compare AC-3_d against AC-7 we have taken results from Bessière, Freuder and Régin as published in [Bessière *et al.*, 1999] and compared them against our own results. Our own algorithms were run on a 1000 Mhz DELL Latitude. To compare our times against Bessière, Freuder and Régin's times, we divided their times by 5 because their experiments were carried out on a 200 MHz Pentium PC [Bessière *et al.*, 1999].

The problem set consists of random problems and Radio Link Frequency Assignment Problems (RLFAPs). The objective for each CSP is that it be made arc-consistent or to decide that this is not possible.

The random CSPs consist of four groups, each of which is uniquely determined by a tuple $\langle\, n, d, p_1, p_2 \,\rangle$. Here, n is the number of variables, d is the (uniform) size of the domains, p_1 is the density of the constraint-graph, and p_2 is the (uniform) tightness of the constraints. Each group contains 50 random CSPs. The four groups that we will consider are given by:

$\langle\, 150, 50, 0.045, 0.500 \,\rangle$ Under-constrained CSPs. To make these problems arc-consistent requires little constraint propagation;

$\langle\, 150, 50, 0.045, 0.940 \,\rangle$ Over-constrained CSPs. To decide that these problems cannot be made arc-consistent requires little constraint propagation;

$\langle\, 150, 50, 0.045, 0.918 \,\rangle$ Low density CSPs at the phase-transition. To make these problems arc-consistent or to decide this is not possible requires much constraint propagation;

$\langle\ \ 50, 50, 1.000, 0.875 \,\rangle$ High density CSPs at the phase-transition. To make these problems arc-consistent or to decide this is not possible requires much constraint propagation.

The RLFAP Problems were obtained from `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks`. To generate the random problems, we used Frost, Dechter, Bessière and Régin's random constraint generator, which is available from `http://www.lirmm.fr/~bessiere/generator.html`. For reproducibility purposes, it should be mentioned that the generator was run with seed 0.

The algorithms that were compared are AC-7 (called AC-7 BFR from here on) as presented in [Bessière *et al.*, 1999], AC-3 (called AC-3 BFR from here on) as presented in [Bessière *et al.*, 1999], our implementation of AC-3, and our implementation of AC-3_d. The arc-heuristic that was used for AC-3 and AC-3_d prefers arc $(\,\alpha, \beta\,)$ to $(\,\alpha', \beta'\,)$ if $s_\alpha < s_{\alpha'}$ or if $s_\alpha = s_{\alpha'} \wedge d_\alpha < d_{\alpha'}$ or $s_\alpha = s_{\alpha'} \wedge d_\alpha = d_{\alpha'} \wedge s_\beta < s_{\beta'}$ or $s_\alpha = s_{\alpha'} \wedge d_\alpha = d_{\alpha'} \wedge s_\beta = s_{\beta'} \wedge d_\beta \leq d_{\beta'}$, where $S_x = |D(x)|$

and $d_x = \deg(x)$. This very expensive heuristic is better for AC-3$_d$ than a lexicographical heuristic with which it almost "degenerates" to AC-3.

It is interesting to mention that a lexicographical arc-heuristic for AC-3$_d$ does not work as well as the more expensive heuristic mentioned before. Always preferring the lexicographically smallest arc (α, β) makes it less likely that (β, α) is also in the queue and this causes AC-3$_d$ to "degenerate" to AC-3.

6.3 Results

Here we shall present and discuss the results for the random and the RLFAP problems.

		$\langle 150, 50, 0.045, 0.500 \rangle$ underconstrained		$\langle 150, 50, 0.045, 0.940 \rangle$ overconstrained	
		checks	time	checks	time
AC-3 BFR		100,010	0.016	514,973	0.074
AC-7 BFR		94,030	0.038	205,070	0.058
AC-3		99,959	0.022	135,966	0.013
AC-3$_d$		50,862	0.019	69,742	0.007
		$\langle 150, 50, 0.045, 0.918 \rangle$ phase-transition/sparse		$\langle 50, 50, 1.000, 0.875 \rangle$ phase-transition/dense	
		checks	time	checks	time
AC-3 BFR	AC	2,353,669	0.338	2,932,326	0.382
	IC	4,865,777	0.734	8,574,903	1.092
AC-7 BFR	AC	481,878	0.154	820,814	0.247
	IC	535,095	0.184	912,795	0.320
AC-3	AC	2,254,058	0.162	4,025,746	0.302
	IC	2,602,318	0.196	6,407,079	0.491
AC-3$_d$	AC	1,734,362	0.140	2,592,579	0.245
	IC	2,010,055	0.171	4,287,835	0.394

Table 6.1: Average Results for Random Problems

The results for the random problems are presented in Table 6.1. The columns "checks" and "time" list the average time and average number of support-checks. For the problems on the phase-transition we have separated the results for problems that could be made arc-consistent and problems which could not be made arc-consistent. The former group is marked by the letters "AC" in the second column and the latter group is marked by the letters "IC" in the second column. We shall first discuss the random problems and then the remaining problems.

For the underconstrained problems AC-3 BFR requires (slightly) more checks than AC-3. That both algorithms require almost the same number of checks is probably caused by the fact that these problems are "almost" arc-consistent so that most arcs have to be checked only once. AC-3 BFR requires less time. This may be caused because it has an arc-heuristic which requires

less overhead than AC-3's. It is difficult to explain the differences between AC-3 BFR and AC-3. Sometimes the former algorithm is better and sometimes the other.

AC-3_d always requires fewer checks than AC-3 BFR and than AC-3. AC-3_d always requires less time than AC-3. Only for the underconstrained problems does AC-3 BFR require less time. For the remaining classes AC-3_d is always better in time. This is consistent with the literature because algorithms which try to be clever by making more inference than AC-3 (BFR) waste time. AC-3 is *not* better in time than AC-3_d. This is caused because its arc-heuristic is the same as that of AC-3_d and because this heuristic requires overhead. The difference between the AC-3 and AC-3_d is mainly caused by AC-3_d having a better domain-heuristic. It is interesting to notice that AC-3_d seems to be a lot better than AC-3 BFR and AC-3 for the overconstrained problems. Apparently AC-3_d is a lot better at detecting such problems.

AC-3_d is better in time and checks than AC-7 BFR for the underconstrained and overconstrained problems. It is interesting that AC-3_d is also much better than AC-7 BFR for the overconstrained problems. For the problems at the phase-transition AC-7 BFR becomes better than AC-3_d in the number of checks. This should not come as a surprise because AC-7 BFR does not repeat checks whereas AC-3_d has to repeat them. The difference in checks is quite significant. AC-3_d performs better in time in the sparse area in the phase-transition region. It performs marginally better in time in the dense area if problems can be made arc-consistent. For the sparse problems in the phase-transition which cannot be made arc-consistent the ratio between the time required by AC-3_d and that required by AC-7 BFR is approximately 1.23 which is significant.

We believe it is fair to say that overall AC-3_d can compete with AC-7 BFR both in time on the wall and checks. Outside the phase-transition AC-3_d performs better than AC-7 BFR both in time and checks. In the phase-transition AC-3_d requires more checks than AC-7 BFR. Only for the sparse problems in the phase-transition should AC-3_d be preferred to AC-7 BFR if one wishes to save time. AC-7 should be preferred for dense problems in the phase-transition.

	AC-3 BFR		AC-7 BFR		AC-3		AC-3_d	
	checks	time	checks	time	checks	time	checks	time
RLFAP#3	615,371	0.050	412,594	0.138	615,371	0.124	267,532	0.092
RLFAP#5	1,735,239	0.126	848,438	0.232	833,282	0.252	250,797	0.136
RLFAP#8	2,473,269	0.168	654,086	0.168	1,170,748	0.420	25,930	0.040
RLFAP#11	971,893	0.072	638,932	0.212	971,893	0.268	406,247	0.186

Table 6.2: Average Results for RLFAP Problems

The results for the RLFAP Problems are presented in Table 6.2. For the RLFAP problems AC-3 BFR performs better than AC-3_d for Problems 3, 5, and 11. This is consistent with our findings for the random problems because these problems are relatively easy. Problems 3 and 11, for example, are already arc-consistent. AC-3_d does significantly better than AC-3 BFR for RLFAP#8 both in time and checks. This is also consistent with our findings for the overconstrained problems because RLFAP#8 cannot be made arc-inconsistent and is relatively easy.

AC-3_d performs better in time and checks than AC-7 BFR for all problems. Again, the results for RLFAP#8 are consistent with our findings for the overconstrained problems. The results for

the other problems are also consistent with the results we found for the random problems because the RLFAP Problems are not in the phase-transition region and are relatively easy.

6.4 Summary

We have presented results from a comparison between AC-3, AC-7, and AC-3_d for random problems and some problems from the Radio Link Frequency Problem Suite. The algorithms had to make a problem arc-consistent or decide that this was impossible.

We have found that AC-3 always requires more support-checks than AC-7 and AC-3_d. Only if problems are easy and underconstrained does AC-3 do better than the other two for as far as time on the wall is concerned.

AC-3_d turns out to be remarkably gifted to cheaply detect overconstrained problems (outside the phase-transition). Except for underconstrained problems, where it requires slightly more time, it always requires fewer checks and less time than AC-3. AC-7 is only better in time for dense problems in the phase-transition. For all other problems AC-3_d performs better in time. AC-7 only requires fewer checks than AC-3_d in the phase-transition region. For the remaining problems AC-3_d requires fewer checks. We believe that our findings demonstrate that AC-3_d can compete with AC-7 both in time on the wall and in the number of support-checks.

Our comparison has been hampered by it being difficult to reproduce results from the literature. It should be interesting to compare the algorithms as part of MAC algorithms. It should also be interesting to compare the algorithms for other classes of random problems. This is something for a future paper.

Chapter 7

Conclusions and Recommendations

In this paper we have presented a general purpose arc-consistency algorithm called AC-3_d whose average time-complexity can compete with AC-7 and whose $\mathbf{O}(e + nd)$ space-complexity improves on AC-7's $\mathbf{O}(ed)$ space-complexity. We have presented experimental results of a comparison between AC-7 and AC-3_d. The results indicate that for the problems under consideration AC-3_d performs better in time on the wall and in the number of support-checks outside the phase-transition. In the phase-transition AC-7 always requires fewer checks. Only for dense problems in the phase-transition region does it require less time.

One reason for the performance of AC-3_d is its double-support domain-heuristic called $\mathcal{D}$. We have compared this heuristic against the lexicographical domain-heuristic $\mathcal{L}$ which is the most commonly used domains-heuristic. Our average time-complexity results have demonstrated beyond doubt that $\mathcal{D}$ is the better heuristic on average.

Our work has been hampered by there being insufficient details in the literature to reproduce experiments. We hope that future papers will contain sufficient information so as to facilitate easy reproducibility. We should like to extend our comparison between AC-3_d and other arc-consistency algorithms.

Bibliography

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers, Inc., San Mateo, California, USA.

[Bessière *et al.*, 1999] C. Bessière, E.G. Freuder, and J.-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[Flajolet and Sedgewick, 1996] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Mellin transform asymptotics. Technical Report Research Report 2956, INRIA, 1996.

[Gaschnig, 1978] 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.

[Gent *et al.*, 1997] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'1997)*, pages 327–340. Springer, 1997.

[Gomes *et al.*, 1997] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 121–135. Springer Verlag, 1997.

[Mackworth and Freuder, 1985] 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.

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

[Sabin and Freuder, 1994] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley & Sons, 1994.

[van Dongen and Bowen, 2000] M.R.C. van Dongen and J.A. Bowen. Improving arc-consistency algorithms with double-support checks. In *Proceedings of the Eleventh Irish Conference on Artificial Intelligence and Cognitive Science (AICS'2000)*, pages 140–149, 2000.

[van Dongen, 2001] M.R.C. van Dongen. A theoretical analysis of the average time-complexity of domain-heuristics for arc-consistency algorithms. In *Proceedings of the Tenth International French Speaking Conference on Logic and Constraint Programming (JFPLC'2001)*, pages 27–41, 2001.

[van Dongen, 2002] M.R.C. van Dongen. *Constraints, Varieties, and Algorithms*. PhD thesis, Department of Computer Science, University College, Cork, Ireland, 2002.

[Wallace and Freuder, 1992] 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.