Lightweight MAC Algorithms

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

Cork Constraint Computation Centre
CS Department
University College Cork
Western Road
Cork
Ireland

Technical Report TR-02-2003

April 2003

Available from http://csweb.ucc.ie/~dongen/papers/4C/03/4C-02-2003.pdf

Abstract

Arc-consistency algorithms are the workhorse of backtrackers that Maintain Arc-Consistency (MAC). This report will provide experimental evidence that, despite common belief to the contrary, it is not always necessary for a good arc-consistency algorithm to have an optimal worst case time-complexity. To sacrifice this optimality allows MAC solvers that (1) do not need additional data structures during search, (2) have an excellent average time-complexity, and (3) have a space-complexity which improves significantly on that of MAC solvers that have optimal arc-consistency components. Results will be presented from an experimental comparison between MAC-2001, MAC-3_d and related algorithms. MAC-2001 has an arc-consistency component with an optimal worst case time-complexity, whereas MAC-3_d does not. MAC-2001 requires additional data structures during search, whereas MAC-3_d does not. MAC-3_d has a space-complexity of $\mathcal{O}(e + nd)$, where n is the number of variables, d the maximum domain size, and e the number of constraints. We shall demonstrate that MAC-2001's space-complexity is $\mathcal{O}(ed \min(n, d))$. MAC-2001 required about 35% more solution time on average than MAC-3_d for easy and hard random problems. MAC-3_d recorded the least solution time for 21 of the 25 real-world problems. Our results indicate that if checks are cheap then lightweight algorithms like MAC-3_d are promising.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Arc-consistency algorithms significantly reduce the size of the search space of Constraint Satisfaction Problems (CSPs) at low costs. They are the workhorse of backtrackers that Maintain Arc-Consistency during search (MAC [Sabin and Freuder, 1994]).

Currently, there seems to be a shared belief in the constraint satisfaction community that, to be efficient, arc-consistency algorithms need an *optimal* worst case time-complexity [Bessière *et al.*, 1995; Bessière and Régin, 2001; Zhang and Yap, 2001]. MAC algorithms like MAC-2001 that have an optimal worst case time-complexity require a space-complexity of at least $\mathcal{O}(ed)$ for creating data structures for remembering their support-checks. We shall prove that MAC-2001's space-complexity is $\mathcal{O}(ed \min(n, d))$ because it has to *maintain* these additional data structures. As usual, n is the number of variables in the CSP, d is the maximum domain size of the variables and e is the number of constraints. We shall present an example illustrating that worst case scenarios for MAC-2001's space-complexity occur for easy CSPs that allow a backtrack free search.

We shall provide evidence to support the claim that good arc-consistency algorithms do not always need an optimal worst case time-complexity. We shall experimentally compare five MAC algorithms. The first algorithm is MAC-2001 [Bessière and Régin, 2001]. MAC-2001's arc-consistency component has an optimal $\mathcal{O}(ed^2)$ worst case time-complexity. The second and third algorithms are MAC-3 and MAC-3_d [Mackworth, 1977; van Dongen, 2003a; 2002a]. The fourth is a new algorithm called MAC-3_p. It lies in between MAC-3 and MAC-3_d. MAC-3, MAC-3_d and MAC-3_p have a better $\mathcal{O}(e + nd)$ space-complexity than MAC-2001 but their arc-consistency components have a non-optimal $\mathcal{O}(ed^3)$ worst case time-complexity. The fifth and last algorithm is MAC-2001_p. It is to MAC-2001 what MAC-3_p is to MAC-3. As part of our presentation we shall introduce some notation to compactly describe ordering heuristics.

For random and real-world problems, for as far as support-checks are concerned, and not to our surprise, MAC-2001_p and MAC-2001 were by far the better algorithms. For any fixed arc-heuristic and for random problems where checks were cheap MAC-3, MAC-3_p and MAC-3_d were *all* better in clock on the wall time than MAC-2001 and MAC-2001_p, with MAC-3_d the best of all. MAC-2001_p required about 21% more time on average than MAC-3_d, whereas MAC-2001 required about 35% more time. For time and solving real-world problems MAC-3_d recorded the least solution time for the vast majority of the problems but MAC-2001 and MAC-3_p

also recorded the best solution times for some of the problems. Things being not as clear as for the random problems it is not possible to say which algorithm should be preferred for the real-world problems that we considered.

This report is the second in a sequence to report about MAC-3 based algorithms with a $\mathcal{O}(e + nd)$ space-complexity that perform significantly better than MAC-2001 for easy *and* hard problems [van Dongen, 2003b]. It is the next in a sequence about MAC-3 based algorithms with a $\mathcal{O}(e+nd)$ space-complexity that are compatible with MAC algorithms that do not repeat checks [van Dongen and Bowen, 2000; van Dongen, 2002b; 2002a; 2003b]. The theoretical principles underlying AC-3_d's double-support heuristic are best described in [van Dongen, 2003a]. To the best of our knowledge the only references to these results have been self-references. Our example about MAC-2001's $\mathcal{O}(ed \min(n, d))$ space-complexity is a first. It points out a weakness of MAC searchers that do not repeat checks.

The results presented in this report are important because of the following. Since the introduction of Mohr and Henderson's AC-4 [Mohr and Henderson, 1986], most work in arc-consistency research has been focusing on the design of better algorithms that do not re-discover (do not repeat checks). This focused research is justified by the observation that, as checks become more and more expensive, there will always be a point beyond which algorithms that repeat will become slower than those that do not and will remain so from then on. However, there are many cases where checks are cheap and it is only possible to avoid re-discoveries at the price of a large additional bookkeeping. To forsake the bookkeeping at the expense of having to re-discover may improve search if checks are cheap *and* if problems become large.

The remainder of this report is organised as follows. Section 2 provides an introduction to constraint satisfaction. Section 3 is an introduction to some notation to describe selection heuristics. Section 4 will describe related work. Section 5 provides a detailed description of the algorithms under consideration and contains a proof that MAC-2001's space-complexity is $\mathcal{O}(ed \min(n, d))$. Section 6 presents our experimental results. Conclusions are presented in Section 7.

Chapter 2

Constraint Satisfaction

A binary *constraint* C_{xy} between variables x and y is a subset of the cartesian product of the domains $D(x)$ of x and $D(y)$ of y. A value $v \in D(x)$ is *supported* by $w \in D(y)$ if $(\, v, w\,) \in C_{xy}$. Similarly, $w \in D(y)$ is supported by $v \in D(x)$ if $(\, v, w\,) \in C_{xy}$.

A *Constraint Satisfaction Problem* (CSP) is a tuple $(\, X, D, C\,)$, where X is a set of variables, $D(\cdot)$ is a function mapping each $x \in X$ to its non-empty domain, and C is a set of constraints between variables in subsets of X. We shall only consider CSPs whose constraints are binary. CSP $(\, X, D, C\,)$ is called *arc-consistent* if its domains are non-empty and for each $C_{xy} \in C$ it is true that every $v \in D(x)$ is supported by y and that every $w \in D(y)$ is supported by x. A *support-check* (consistency-check) is a test to find out if two values support each other.

The *tightness* of the constraint C_{xy} between x and y is defined as $1 - |\, C_{xy}\,|/|\, D(x) \times D(y)\,|$, where $\cdot \times \cdot$ denotes cartesian product. The *density* of a CSP is defined as $2e/(n^2 - n)$, for $n > 1$.

The *(directed) constraint graph* of CSP $(\, X, D, C\,)$ is the directed graph whose nodes are given by X and whose arcs are given by $\cup_{C_{xy} \in C} \{\, (\, x, y\,), (\, y, x\,)\,\}$. The *degree* of a variable in a CSP is the number of neighbours of that variable in the (directed) constraint graph of that CSP.

MAC is a backtracker that maintains arc-consistency during search. MAC-i uses arc-consistency algorithm AC-i to maintain arc-consistency.

The following notation is not standard but will turn out useful. Let $\delta_o(v)$ be the original degree of v, let $\delta_c(v)$ be the current degree of v, let $\kappa(v) = |\, D(v)\,|$, and let $\#(v)$ be a *unique* number which is associated with v. We will assume that $\#(v) \leq \#(w)$ if and only if v is lexicographically less than or equal to w.

Chapter 3

Operators for Selection Heuristics

In this chapter we shall introduce notation to describe and "compose" variable and arc selection heuristics. The reader not interested in the nitty gritty details of such heuristics may wish to skip this chapter and return to it later. The notation was introduced in [van Dongen, 2003b, Chapter 3] but to make this report self-contained the main points are repeated here. Motivation, a more detailed presentation, and more examples may be found in [van Dongen, 2003b, Chapter 3].

It is recalled that a relation on set T is called a *quasi-order* on T if it is reflexive and transitive. A relation, $\prec$, on T is called *linear* if $v \prec w \vee w \prec v$ for all v, $w \in T$. Linear quasi-orders may allow for "ties," i.e. they may allow for situations where $v \prec w \wedge w \prec v \wedge v \neq w$. A quasi-order $\preceq$ is called a *partial order* if $v \preceq w \wedge w \preceq v \implies v = w$ for all v, $w \in T$. An *order* (also called a *linear order*) is a partial order that is also a linear quasi-order. An order $\preceq$ *prefers* v to w if and only if $v \preceq w$.

The *composition* of order $\preceq_2$ and linear quasi-order $\preceq_1$ is denoted $\preceq_2 \bullet \preceq_1$. It is the unique order on T which is defined as follows:

$$v \preceq_2 \bullet \preceq_1 w \iff (v \preceq_1 w \wedge \neg w \preceq_1 v) \vee (v \preceq_1 w \wedge w \preceq_1 v \wedge v \preceq_2 w)\,.$$

In words, $\preceq_2 \bullet \preceq_1$ is the selection heuristic that uses $\preceq_1$ and "breaks ties" using $\preceq_2$. Composition associates to the left, i.e. $\preceq_3 \bullet \preceq_2 \bullet \preceq_1$ is equal to $(\preceq_3 \bullet \preceq_2) \bullet \preceq_1$.

Let $\preceq$ be a linear quasi-order on T, and let $f :: Y \mapsto T$ be a function. Then $\otimes^f_\preceq$ is the unique linear quasi-order on Y which is defined as follows:

$$v \otimes^f_\preceq w \iff f(v) \preceq f(w)\,, \qquad \text{for all } v, w \in Y\,.$$

Finally, let $\pi_i((\,v_1, \ldots, v_n\,)) = v_i$ for $1 \leq i \leq n$.

We are now in a position where we need no more notation. For example, the minimum domain size heuristic with a lexicographical tie breaker is given by $\otimes^\#_\leq \bullet \otimes^\kappa_\leq$, the ordering on the maximum original degree with a lexicographical tie breaker is given by $\otimes^\#_\leq \bullet \otimes^{\delta_o}_\geq$, the *Brelaz heuristic* (cf. [Gent *et al.*, 1996]) with a lexicographical tie breaker is given by $\otimes^\#_\leq \bullet \otimes^{\delta_c}_\geq \bullet \otimes^\kappa_\leq$, and $\otimes^{\#\circ\pi_2}_\leq \bullet \otimes^{\#\circ\pi_1}_\leq$ is the lexicographical arc-heuristic. As usual, $\cdot \circ \cdot$ denotes function composition.

Chapter 4

Related Literature

In 1977, Mackworth presented an arc-consistency algorithm called AC-3 [Mackworth, 1977]. AC-3 has a $\mathcal{O}(ed^3)$ bound for its worst case time-complexity [Mackworth and Freuder, 1985]. AC-3 has a $\mathcal{O}(e + nd)$ space-complexity. AC-3 cannot remember all its support-checks. Pseudo code for AC-3 and for the $revise$ algorithm upon which it depends is depicted in Figures 4.1 and 4.2. The "foreach $s \in S$ do $statement$" construct assigns the members in S to s from small to big and carries out $statement$ after each assignment. AC-3 uses *arc-heuristics* to repeatedly select and remove an arc, $(\,x, y\,)$, from a data structure called a *queue* (a set, really) and to use the constraint between x and y to *revise* the domain of x. Here, to revise the domain of x using the constraint between x and y means to remove the values from $D(x)$ that are not supported by y. AC-3's arc-heuristics determine the constraint that will be used for the next support-check. Besides these arc-heuristics there are also *domain-heuristics*. These heuristics, if given the constraint that will be used for the next support-check, determine the values that will be used for the next support-check. The interested reader is referred to [Mackworth, 1977; Mackworth and Freuder, 1985] for further information about AC-3.

```
function AC-3( X ) : Boolean;
    Q := { ( x, y ) ∈ X² : x and y are neighbours };
    while Q ≠ ∅ do begin
        select and remove any arc ( x, y ) from Q;
        if not revise(x, y, change_x) then
            return false;
        else if change_x then
            Q := Q ∪ { ( z, x ) : z ≠ y, z is a neighbour of x };
    end;
    return true;
end;
```

Figure 4.1: The AC-3 algorithm.

```
function revise(x, y, var change_x) : Boolean;
begin
    change_x := false;
    foreach r ∈ D(x) do
        if ∄c ∈ D(y) s.t. c supports r then begin
            D(x) := D(x) \ { r };
            change_x := true;
        end;
    return D(x) ≠ ∅;
end;
```

Figure 4.2: Algorithm $revise$.

Wallace and Freuder pointed out that arc-heuristics can influence the efficiency of arc-consistency algorithms [Wallace and Freuder, 1992]. Similar observations were made by Gent *et al.* [Gent *et al.*, 1997]. Despite these findings only a few authors describe the heuristics that were used for their experiments. We believe that to facilitate ease of replication all information to repeat experiments should be described in full. This includes information about arc-heuristics.

Bessière and Régin presented AC-2001, which is based on AC-3 [Bessière and Régin, 2001] (see also [Zhang and Yap, 2001] for a similar algorithm). AC-2001 revises one domain at a time. The main difference between AC-3 and AC-2001 is that AC-2001 uses a lexicographical domain-heuristic and that for each variable x, for each $v \in D(x)$ and each constraint between x and another variable y it remembers the last support for $v \in D(x)$ with y so as to avoid repeating checks that were used before to find support for $v \in D(x)$ with y. AC-2001 has an optimal upper bound of $\mathcal{O}(ed^2)$ for its worst case time-complexity and its space-complexity is $\mathcal{O}(ed)$. AC-2001 behaves well on average. It was observed that AC-3 is a good alternative for stand alone arc-consistency if checks are cheap and CSPs are under-constrained but that AC-3 is very slow for over-constrained CSPs and CSPs in the phase transition [Bessière *et al.*, 1999; Bessière and Régin, 2001].

We made similar observations in experimental comparisons between AC-7, AC-2001 and AC-3_d, which is a cross-breed between Mackworth's AC-3 and Gaschnig's DEE [Mackworth, 1977; Gaschnig, 1978; van Dongen, 2002a]. We did *not* consider search. The only difference between AC-3 and AC-3_d is that AC-3_d sometimes takes two arcs out of the queue and simultaneously revises *two* domains with a *double-support* domain-heuristic. A double-support heuristic is a heuristic that prefers checks between two values each of whose support statuses are unknown. For two-variable CSPs the double-support heuristic is optimal and requires about half the checks that are required by a lexicographical heuristic if the domain sizes of the variables are about equal and sufficiently large [van Dongen, 2003a]. AC-3_d and MAC-3_d have a low $\mathcal{O}(e + nd)$ space-complexity. Our results indicated that AC-3_d was promising for stand alone arc-consistency.

Chapter 5

Detailed Description of Other MAC-3 Based Algorithms

5.1 Introduction

In this chapter we shall describe MAC-3_d, MAC-3_p, MAC-2001, and MAC-2001_p in more detail. The presentation is to provide a good understanding of the basic machinery of the algorithms and to highlight the differences between them. The presentation is not meant to describe an efficient implementation. As part of the presentation we shall prove that MAC-2001 has a $\mathcal{O}(ed\min(n,d))$ space-complexity.

5.2 MAC-3_d and MAC-3_p

AC-3_d is a cross-breed between AC-3 and DEE [Mackworth, 1977; Gaschnig, 1978]. Pseudo-code for AC-3_d is depicted in Figure 5.1. The only difference between AC-3 and AC-3_d is that AC-3_d sometimes takes two arcs out of the queue and *simultaneously* revises *two* domains with Algorithm $\mathcal{D}$. Pseudo code for $\mathcal{D}$ is depicted in Figure 5.2. $\mathcal{D}$ uses a *double-support* domain-heuristic, i.e. a heuristic which prefers double-support checks. The constants $unsupported$, $single$, and $double$ that are used in $\mathcal{D}$ are pairwise different and smaller than the values in the domains of the variables.

AC-3_p is a "poor man's" version of AC-3_d; It is not as efficient but easier to implement. It can be obtained from AC-3_d by replacing the call to $\mathcal{D}$ in the 7$^{\text{th}}$ line of AC-3_d by "$revise(x, y, change_x)$ and $revise(y, x, change_y)$." The difference between AC-3_p and AC-3_d is AC-3_d's double-support heuristic.

$\mathcal{D}$'s space-complexity is $\mathcal{O}(d)$. In $\mathcal{D}$ the *row-support* are the values in $D(x)$ that are supported by y and the *column-support* are the values in $D(y)$ that are supported by x. It easy to prove that $\mathcal{D}$ correctly computes its row-support. To prove that it also correctly computes its column-support is not much more difficult. The proof relies on the fact that after establishing row-support any c that is not yet known to be supported can only be supported by an $r \in D(x)$ such that $rkind[r] = double \land rsupp[r] < c$.

```
function AC-3_d( X ) : Boolean;
    Q := { ( x, y ) ∈ X^2 : x and y are neighbours };
    while Q ≠ ∅ do begin
        select and remove any arc ( x, y ) from Q;
        if ( y, x ) is also in Q then begin
            remove ( y, x ) from Q;
            if not 𝒟(x, y, change_x, change_y) then
                return false;
            else begin
                if change_x then
                    Q := Q ∪ { ( z, x ) : z ≠ y, z is a neighbour of x };
                if change_y then
                    Q := Q ∪ { ( z, y ) : z ≠ x, z is a neighbour of y };
            end
        end
        else if not revise(x, y, change_x) then
            return false;
        else if change_x then
            Q := Q ∪ { ( z, x ) : z ≠ y, z is a neighbour of x };
    end;
    return true;
end;
```

Figure 5.1: The AC-3_d algorithm.

```
function 𝒟(x, y, var change_x, var change_y) : Boolean;
begin
    change_x := false;
    change_y := false;
    foreach c ∈ D(y) do
        csupp[c] := unsupported;
    /* Compute row-support. */
    foreach r ∈ D(x) do begin
        rkind[r] := unsupported;
        rsupp[r] := unsupported;
        if ∃c ∈ D(y) s.t. csupp[c] = unsupported and c supports r then begin
            rsupp[r] := first such value c;
            csupp[rsupp[r]] := r;
            rkind[r] := double;
        end
        else if ∃c ∈ D(y) s.t. csupp[c] ≠ unsupported and c supports r then begin
            rsupp[r] := first such value c;
            rkind[r] := single;
        end
        else begin
            D(x) := D(x) \ { r };
            change_x := true;
        end;
    end;
    /* Complete column-support. */
    foreach c ∈ D(y) s.t. csupp[c] = unsupported do
        if ∄r ∈ D(x) s.t. rsupp[r] < c and rkind[r] = double and r supports c then begin
            D(y) := D(y) \ { c };
            change_y := true;
        end;
    return D(x) ≠ ∅;
end;
```

Figure 5.2: Algorithm $\mathcal{D}$.

AC-3$_d$ and AC-3$_p$ inherit their $\mathcal{O}(ed^3)$ worst case time-complexity and $\mathcal{O}(e+nd)$ space-complexity from AC-3. MAC-3$_d$ (MAC-3$_p$) is implemented by replacing AC-3 in MAC-3 by AC-3$_d$ (AC-3$_p$). The space-complexity of MAC-3$_d$ and MAC-3$_p$ is equal to $\mathcal{O}(e + nd)$.

5.3 MAC-2001 and MAC-2001$_p$

Pseudo-code for an arc-based version of AC-2001 and the *revise*-2001 algorithm upon which it depends is depicted in Figures 5.3 and 5.4. For the purpose of the presentation of AC-2001 it is assumed that the values in the domains are ordered from small to big. For each variable x, for each value $v \in D(x)$, and for each neighbour y of x it is assumed that $last[x][v][y]$ is initialised to some value that is smaller than the values in $D(y)$.

```
function AC-2001( X ) : Boolean;
begin
   Q := { ( x, y ) ∈ X² : x and y are neighbours };
   while Q ≠ ∅ do begin
      select and remove any arc ( x, y ) from Q;
      if not revise-2001(x, y, change_x) then
         return false;
      else if change_x then
         Q := Q ∪ { ( z, x ) : z ≠ y, z is a neighbour of x };
   end;
   return true;
end;
```

Figure 5.3: Arc-based version of AC-2001.

```
function revise-2001( x, y, var change ) : Boolean;
begin
   change := false;
   foreach r ∈ D(x) do
      if last[x][r][y] ∉ D(y) then
         if ∃c ∈ D(y) s.t. c > last[x][r][y] and c supports r then
            last[x][r][y] := the first such value c;
         else begin
            D(x) := D(x) \ { r };
            change := true;
         end;
   return D(x) ≠ ∅;
end;
```

Figure 5.4: *revise*-2001.

AC-2001 finds support for $v \in D(x)$ with y by checking against the values in $D(y)$ from small to large. It uses a counter $last[x][v][y]$ to record the last check that was carried out. This allows it to save checks the next time support for $v \in D(x)$ has to be found with y if $last[x][v][y] \in D(y)$. Furthermore, checks are saved by not looking for support with values that are less than or equal to $last[x][v][y] \in D(y)$.

MAC-2001 requires additional data structures during search. It maintains the counter $last[x][v][y]$ to remember the last support for $v \in D(x)$ with $D(y)$. The space-complexity of *last* is $\mathcal{O}(ed)$ [Bessière and Régin, 2001]. It seems to have gone unnoticed so far that MAC-2001 has a $\mathcal{O}(ed \min(n, d))$ space-complexity. The reason for this space-complexity is that MAC-2001 has to *maintain* the data structure $last$. This only seems to be possible using one of the following two methods (or a combination):

1. Save all relevant counters once before AC-2001. Upon backtracking these counters have to be restored. This requires a $\mathcal{O}(ned)$ space-complexity because $\mathcal{O}(ed)$ data structures may have to be saved n times.

2. Save each counter before the assignment to $last[x][v][y]$ in *revise*-2001 and count the number of changes, c, that were carried out. Upon backtracking, restore the c counters in the

reverse order. This comes at the price of a space-complexity of $\mathcal{O}(ed^2)$ because each of the $2ed$ counters may have to be saved $\mathcal{O}(d)$ times.

Therefore, MAC-2001's space-complexity is $\mathcal{O}(ed \min(n, d))$. Christian Bessière (private communication) implemented MAC-2001 using Method 2.

The consequences of MAC-2001's space requirements can be prohibitive. For example, without loss of generality we may assume the usual lexicographical value ordering. Let $n = d > 1$ and consider the binary CSP where all variables should be pairwise different. Finally, assume that Method 2 is used for MAC-2001 (Method 1 will lead to a similar order of space-complexity). Note that the "first" solution can be found with a backtrack free search. Also note that in the first solution i is assigned to the i-th variable. We shall see that MAC-2001 will require a lot of space to solve the given CSP.

Just before the assignment of i to the i-th variable we have the following. For each variable x, for each variable $y \neq x$, and for each $v \in D(x) = \{ i, \ldots, n \}$ we have $last[x][v][y] = \min(\{ i, \ldots, n \} \setminus \{ v \})$. To make the CSP arc-consistent after the assignment of i to the i-th current variable, (only) the value i has to be removed from the domains of the future variables. Unfortunately, for each of the remaining $n - i$ future variables x, for each of the remaining $n - i$ values $v \in D(x) \setminus \{ i \}$, and for each of the remaining $n - i - 1$ future variables $y \neq x$, i was the last known support for $v \in D(x)$ with y. This means that $(n - i)^2 \times (n - i - 1)$ counters must be saved and incremented during the AC-2001 call following the assignment of i to the i-th variable. In total, MAC-2001 has to save $\sum_{i=1}^{n}(n - i)^2 \times (n - i - 1)$, i.e. $(n - 2) \times (n - 1) \times n \times (3n - 1)/12$ counters. For $n = d = 500$, MAC-2001 will require space for at least $15{,}521{,}020{,}750$ counters and this may not be available on every machine. Sometimes MAC algorithms that do not re-discover *do* require a lot of space, even for deciding relatively small CSPs that allow a backtrack free search.

The last thing that remains to be done in this chapter is to describe AC-2001$_p$. This algorithm is to AC-2001 what AC-3$_p$ is to AC-3. If its arc-heuristic selects (x, y) from the queue and if (y, x) is also in the queue then it will remove both and use (at most) two calls to $revise$-2001 to revise the domains of x and y.

Chapter 6

Experimental Results

6.1 Introduction

In this chapter we shall experimentally compare MAC-2001, MAC-2001$_p$, MAC-3$_d$, MAC-3$_p$ and MAC-3 for random and real-world problems. For the random problems we implemented support-checks as cheap lookup operations in arrays. For the real world problems we implemented support-checks as (more) expensive function calls.

6.2 Implementation Details

All implementations were based on our own implementation of MAC-3$_d$ and all used the same basic data structures that were used by MAC-3$_d$. The implementations of MAC-2001 and MAC-3$_p$ were arc-based. This allowed us to evaluate the algorithms for different arc-heuristics. Previously, we used Christian Bessière's variable based implementation of MAC-2001 [van Dongen, 2003b]. However, Bessière's implementation came with only one arc-heuristic, was a specialised version for random problems and was about 17% slower than our own implementation.

All solvers were real-full-look-ahead solvers and to ensure that they visited the same nodes in the search tree they were equipped with the same dom/deg variable ordering heuristic. Using the notation introduced in Section 3 this heuristic is given by $\otimes^{\#}_{\leq} \bullet \otimes^{f}_{\leq}$, where $f(v) = \kappa(v)/\delta_o(v)$. We considered three different arc-heuristics, called lex, $rlex$, and $comp$. Using the notation introduced in Section 3 these can be defined as:

$$
\begin{aligned}
lex &= \otimes^{\#\circ\pi_2}_{\leq} \bullet \otimes^{\#\circ\pi_1}_{\leq}\,, \\
rlex &= \otimes^{\#\circ\pi_1}_{\leq} \bullet \otimes^{\#\circ\pi_2}_{\leq}\,, \quad \text{and} \\
comp &= \otimes^{\#\circ\pi_2}_{\leq} \bullet \otimes^{\delta_c\circ\pi_2}_{\geq} \bullet \otimes^{\kappa\circ\pi_2}_{\leq} \bullet \otimes^{\#\circ\pi_1}_{\leq} \bullet \otimes^{\delta_c\circ\pi_1}_{\geq} \bullet \otimes^{\kappa\circ\pi_1}_{\leq}\,.
\end{aligned}
$$

The queue was implemented as a directed graph G. This data structure contains a $\mathcal{O}(n)$ linked list N to represent the nodes of G that have an incoming arc: $N = \{\, x : (x, y) \in G \,\}$. The data structure also contains a $\mathcal{O}(n)$ array that contains a linked lists for each member of N to represent the other ends of the arcs. The total size of these linked lists does not exceed

$\mathcal{O}(e)$. This brings the space-complexity for our queue representation to $\mathcal{O}(e)$. We did not use a $\mathcal{O}(n \times n)$ lookup table to quickly find out if a certain arc was in the queue. Had we used such table then we should have changed our claim about the space-complexity of MAC-3, MAC-3_p and MAC-3_d to $\mathcal{O}(n \max(n, d))$.

With this implementation of the queue, selecting the best arc with respect to lex takes $\mathcal{O}(1)$ time, whereas selecting the best arc with respect to $rlex$ and $comp$ takes $\mathcal{O}(n)$ time. The heuristic $comp$ requires a few more words. At the moment of writing is the best known arc-heuristic for MAC-3_d. Further in this chapter we shall see that it is also an excellent heuristic for the remaining algorithms. Profiling revealed that arc-selection for MAC-3_d with $comp$ usually takes between 10% and 20% of the solution time, whereas selection with lex hardly takes any time. However, $comp$ has a far better effect on constraint propagation than both lex and $rlex$ and investing in it is well spent. We intend to cut down the time for arc-selection with $comp$ by supporting it with a special data type for the queue. It is not quite clear *why* this heuristic has such a good effect on constraint propagation. This is something we intend to investigate further.

6.3 Random Problems

Random problems were generated for $15 \leq n = d \leq 30$. We will refer to the class of problems for a given combination of $n = d$ as the problem class with *size* n. The problems were generated as follows. For each problem size and each combination $(\,C, T\,)$ of average density C and uniform tightness T in $\{\,(\,i/20, j/20\,) \,:\, 1 \leq i, j \leq 19\,\}$ we generated 50 random CSPs. Next we computed the average number of checks and the average time that was required for deciding the satisfiability of each problem using MAC search. All problems were run to completion. Frost *et al.*'s model B [Gent *et al.*, 2001] random problem generator was used to generate the problems (`http://www.lirmm.fr/~bessiere/generator.html`).

The test was carried out in parallel on 50 identical machines. All machines were Intel Pentium III machines, running SuSe Linux 8.0, having 125 MB of RAM, having a 256 KB cach size, and running at a clock speed of about 930 MHz. Between pairs of machines there were small (less than 1%) variations in clock speed. Each machine was given a unique identifier in the range from 1 through 50. For each machine random problems were generated for each combination of density and tightness. The CSP generator on a particular machine was started with the seed given by 1000 times the machine's identifier. All problems fitted into memory and no swapping occurred. *The solution time* included *counting the checks*. The total time for our comparison is equivalent to more than 100 days of computation on a single machine.

Figures 6.1 and 6.2 depict scatter plots of the time required by MAC-2001_p and MAC-3_d both equipped with a $comp$ heuristic versus the number of checks that they required to find the first solution for problem size 30. Both figures suggest that the solution time linearly depends on the number of checks. A similar linear relationship between the solution time and the number of checks was observed for all algorithms, for all heuristics, and *all* problem sizes. Note that the figures demonstrate that many problems were difficult and took tens of minutes to hours to complete.

Figure 6.3 depicts a scatter plot of the checks required by MAC-3_d with $comp$ versus the

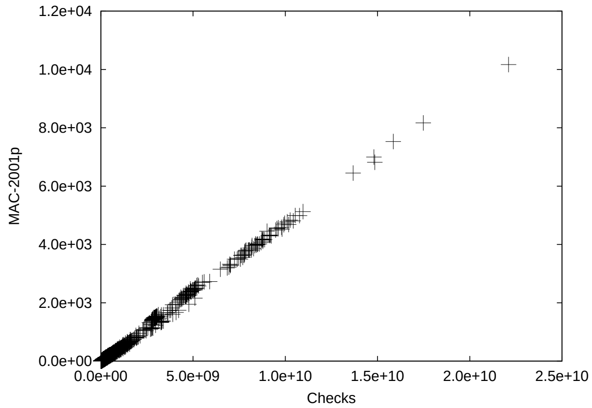

Figure 6.1: Size 30: Scatter plot of time of MAC-2001$_p$ with $comp$ arc-heuristic for first solution vs. average number of checks.

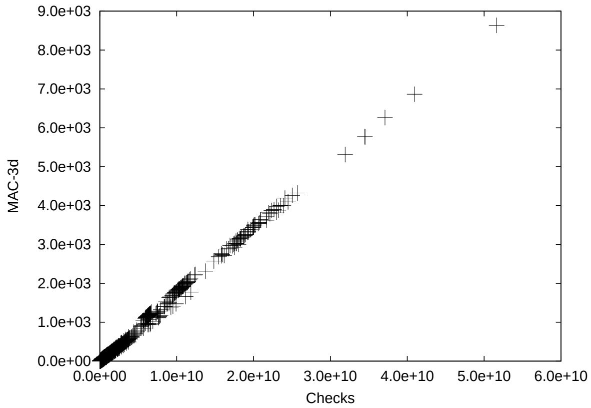

Figure 6.2: Size 30: Scatter plot of time of MAC-3$_d$ with $comp$ arc-heuristic for first solution vs. average number of checks.

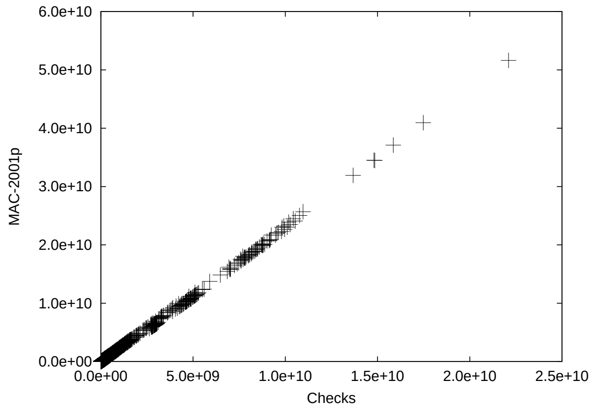

Figure 6.3: Size 30: Scatter plot of number checks of MAC-2001$_p$ with $comp$ arc-heuristic for first solution vs. number checks of MAC-3$_d$ with $comp$ arc-heuristic.

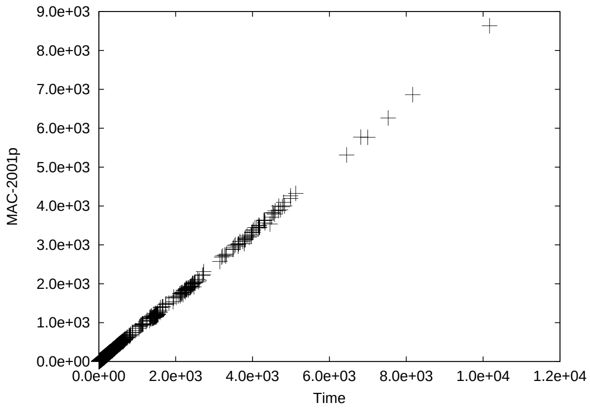

Figure 6.4: Size 30: Scatter plot of time of MAC-2001$_p$ with $comp$ arc-heuristic for first solution vs. time of MAC-3$_d$ with $comp$ arc-heuristic.

number of checks required by $\mathrm{MAC}\text{-}2001_p$ with $comp$ for problem size 30. Figure 6.4 depicts a scatter plot of the time required by $\mathrm{MAC}\text{-}3_d$ with $comp$ versus the time required by $\mathrm{MAC}\text{-}2001_p$ with $comp$ for problem size 30. Both figures suggest that there is a linear relationship between the number of checks required by $\mathrm{MAC}\text{-}3_d$ and $\mathrm{MAC}\text{-}2001_p$ and between the solution times of $\mathrm{MAC}\text{-}3_d$ and $\mathrm{MAC}\text{-}2001_p$. Again, similar linear relationships were observed for other combinations of algorithms.

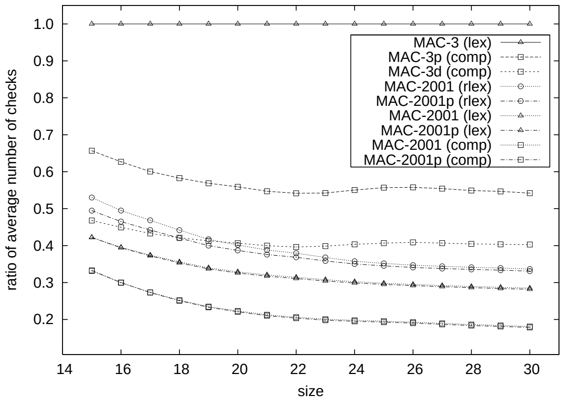

Figure 6.5: Ratio of average number of checks vs. problem size for random problems and search. For each size the average number checks is divided by the average number of checks required by MAC-3 with the lex arc-heuristic.

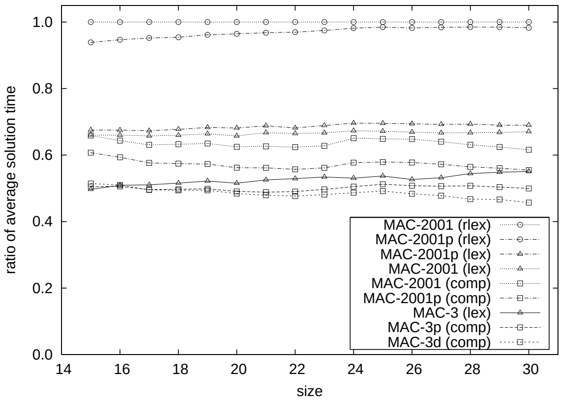

Figure 6.6: Ratio of average solution time vs. problem size for random problems and search. For each size the average time is divided by the average time required by MAC-2001 with the $rlex$ arc-heuristic.

Figure 6.5 depicts the ratio between the average number of checks on the one hand and the average number of MAC-3 with a lex arc-heuristic on the other for problem sizes 15–30 and different combinations of algorithms and arc-heuristics. Similarly, Figure 6.6 depicts the ratio between the average solution time and the average solution time of MAC-2001 with an $rlex$ arc-heuristic. The order from top to bottom in which the algorithms and heuristics are listed in the legends of the figures corresponds to the height of their graphs for problem size 30. It is difficult to see but what seem to be two lines at the bottom of Figure 6.5 are two pairs of lines. The pair at the bottom corresponds to MAC-2001 and $\mathrm{MAC}\text{-}2001_p$ with a $comp$ heuristic. The other pair corresponds to MAC-2001 and $\mathrm{MAC}\text{-}2001_p$ with a lex heuristic. As the problem size increases the lines for MAC-2001 and $\mathrm{MAC}\text{-}2001_p$ with an $rlex$ heuristic also seem to converge. $\mathrm{MAC}\text{-}2001_p$ and MAC-2001 with a $comp$ heuristic are the best when it comes to saving checks.

It is interesting to observe that when MAC-3 and $\mathrm{MAC}\text{-}3_p$ are both equipped with the same heuristic then $\mathrm{MAC}\text{-}3_p$ solves more quickly on average (except for small problem sizes). Similarly, $\mathrm{MAC}\text{-}2001_p$ solves more quickly than MAC-2001. Apparently, the choice to process the reverse arc if it is in the queue has a good effect on constraint propagation. It is our intention to further investigate the effect that always processing the reverse arc if it is also in the queue has on constraint propagation.

For problem size 30 the average solution time of MAC-2001$_p$ was about 36.289 seconds, that of MAC-2001 was about 40.294 seconds, and that of MAC-3$_d$ was about 29.910 seconds. On average and over all problems MAC-2001$_p$ required about 21% more time than MAC-3$_d$, whereas MAC-2001 required about 35% more time.

For any heuristic that was under consideration and when it comes to saving time MAC-2001 and MAC-2001$_p$ are never better on average than MAC-3, MAC-3$_p$ and MAC-3$_d$. Our findings about MAC-3$_d$ are consistent with our previous work [van Dongen, 2002a; 2003b] The results about MAC-2001 and MAC-3 are in contrast with other results from the literature [Bessière and Régin, 2001]. However, our findings should not be dismissed because the literature does not agree; Our testing has been fair and thorough and we cannot recall having seen such comprehensive comparison before. MAC-3 with lex requires about 5 times more checks on average than MAC-2001 and MAC-2001$_p$ with $comp$ but solves more quickly on average. The lack of intelligence in it's strategy for propagation does not seem to hinder MAC-3 at all when checks are cheap. It is even more interesting because MAC-3 performed even better with $comp$.

Figures 6.5 and 6.6 seem to suggest that as a rule and given one of the algorithms MAC-2001 and MAC-2001$_p$ the heuristic $comp$ was better than lex which, in its turn, was better than $rlex$ both for checks and time. A further investigation of the test data revealed that this was, indeed, true.

For random problems and for clock on the wall time the best algorithm was MAC-3$_d$ with a $comp$ heuristic. MAC-3$_p$ with a $comp$ arc-heuristic was a good second. MAC-3$_d$'s double-support heuristic allows it to improve on MAC-3$_p$. Overall, the best algorithm from the MAC-2001 family required more than 21% more time on average than MAC-3$_d$.

6.4 Statistical Analysis

In this section we shall statistically analyse the relationship between the average solution time of MAC-2001$_p$, the quickest algorithm from the MAC-2001 family, and MAC-3$_d$, the quickest lightweight, both with a $comp$ arc-heuristic. In the remainder of this section, let $\mathcal{T}_i$ be the average time required by MAC-i.

To find out about the relationship between $\mathcal{T}_{2001_p}$ and $\mathcal{T}_{3_d}$ we used a standard linear regression analysis for the model $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times \text{tightness} + d \times \text{density}$ for problem sizes 15–30. Figures 6.7–6.10 depict the values for the coefficients a–d versus the problem size. The error bars in the figures indicate the size of the 95% confidence intervals of these coefficients.

Figure 6.11 depicts the ratio between the model sum of squares and the sum of squares values (usually denoted R^2). R^2 was always at least 0.98. When the problem size increased this R^2 value tended to converge quickly to 1. This indication that the models are good and became better as the problem size increased.

Figure 6.8 suggests that the ratio $\mathcal{T}_{2001_p}/\mathcal{T}_{3_d}$ increases as the problem size increases. If this is a real trend then not only will MAC-2001$_p$ require more time than MAC-3$_d$ but also will it require proportionally more time as problems become more and more difficult.

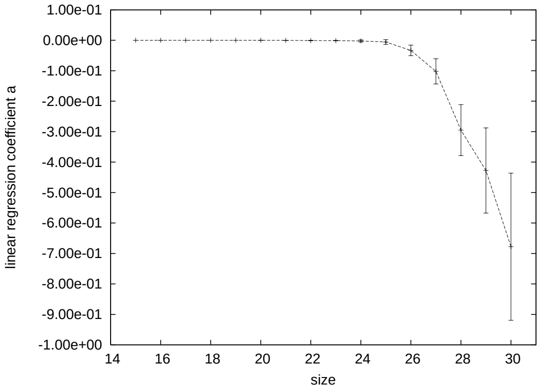

Figure 6.7: Value of a for $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times tightness + d \times density$ for search.

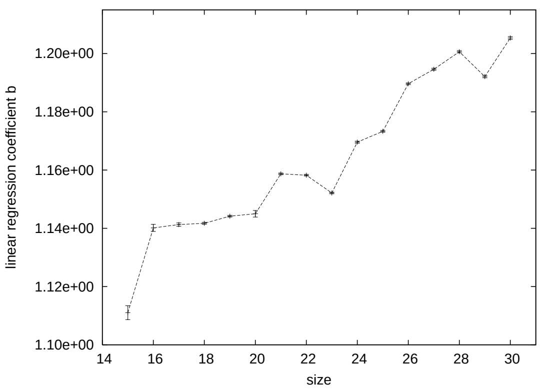

Figure 6.8: Value of b for $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times tightness + d \times density$ vs. size.

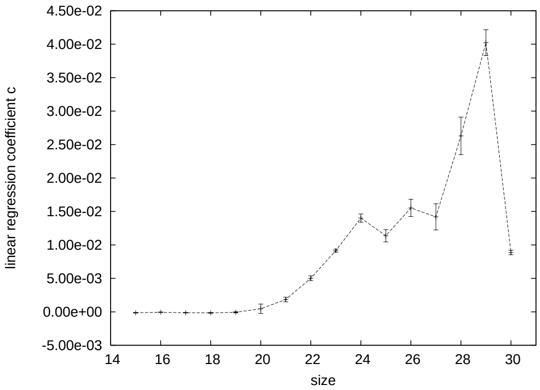

Figure 6.9: Value of c for $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times tightness + d \times density$ vs. size.

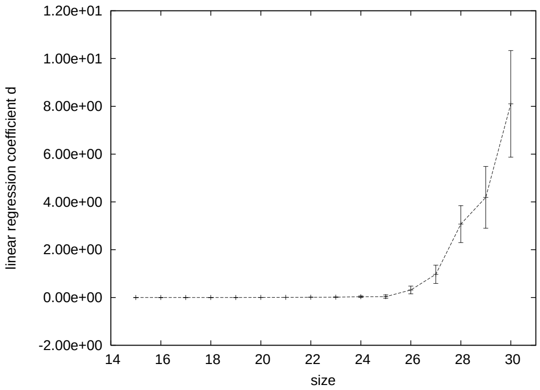

Figure 6.10: Value of d for $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times tightness + d \times density$ vs. size.

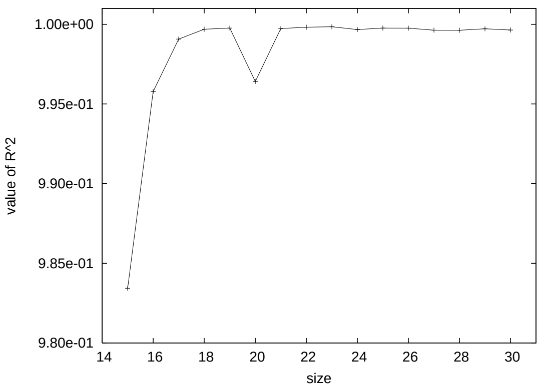

Figure 6.11: Value of R^2 for $\mathcal{T}_{2001_p} = a + b \times \mathcal{T}_{3_d} + c \times tightness + d \times density$ vs. size.

6.5 Real-World Problems

The real-world problems came from the CELAR suite [CELAR, 1994]. We considered all problems (RLFAP 1–11 and GRAPH 1–14). However, we did not consider optimisation but only considered satisfiability.

These problems have become a sort of a standard benchmark for real-world problems. All constraints in these problems are of the form $|\,v-w\,| \leq c$, $|\,v-w\,| \geq c$ $|\,v-w\,| < c$, $|\,v-w\,| > c$ or $|\,v-w\,| = c$ for different coefficients c. Checks were implemented as function calls. We used the following implementation in C.

```
#define SATISFIES( /* CONSTRAINT_POINTER */ constraint \
                 , /* int */ val_v                    \
                 , /* int */ val_w                    \
                 )                                    \
  constraint->rlfap_function( constraint->idx_a,       \
                              constraint->idx_b,       \
                              val_v,                   \
                              val_w,                   \
                              constraint->rlfap_val )
```

Here `constraint->idx_a` and `constraint->idx_b` are the numbers of the variables that are involved in the constraint and `constraints->rlfap_val` stores the coefficient c. Note that the function call requires some overhead. The overhead consists of the retrieval of the numbers of the variables and that of the coefficient c. For each comparison operator $\leq$, $\geq$, $<$, $>$, and $=$ a function was implemented to compare $|\,v-w\,|$ and c. For example, the function to decide if $|\,v-w\,| = c$ was implemented as follows:

```
int rlfap_eq( int idx_a, int idx_b, int val_a, int val_b, int number ) {
  return ( abs( domains[ idx_a ].numbers[ val_a ] -
                domains[ idx_b ].numbers[ val_b ] ) == number);
}
```

and for each constraint of the form $|\,v-w\,| = c$ between the variables with numbers `idx_a` and `idx_b` we initialised the constraint "`constraint`" between these variables as follows:

```
/* initialises ''constraint'' between variables a and b with numbers idx_a and idx_b
 * such that for each v in Domain( a ) and w in Doman( b ) we have | v - w | == c.
 */
constraint->idx_a = idx_a;
constraint->idx_b = idx_b;
constraint->rlfap_val = c;
constraint->rlfap_function = rlfap_eq;
```

Notice that the function `rlfap_eq` requires 4 array subscriptions, one more function call to a function called `abs`, two "offset" computations (the `.numbers` operations), and a comparison. It is hoped that the reader agrees that for the real-world problems the checks are more difficult than for the random CSPs where they only required two array subscriptions. Note that the solution time *excludes* counting the checks. Comments about this different set up will be provided at the end of this section.

All problems were solved on the same machine. This machine was one of the machines that were used to solve the random problems. The specifications of such machine can be found in Section 6.3. For every problem we computed the average solution time over 50 runs. For all problems $d = 44$.

The results for the tests are depicted in Tables 6.1–6.6 on Pages 23–28. For each problem the least average number of checks and least average solution time recorded for that problem for all arc-heuristics are printed in **green and bold face**. For each of the remaining heuristics the least average number of checks and least average solution time are printed in *blue and italics*. The values in column "ac" tell whether the problem could be made arc-consistent initialy.

Again MAC-2001 and MAC-2001$_p$ are the best when it comes to saving checks. It pays off for RLFAP 11, GRAPH 4, GRAPH 6, GRAPH 10, and GRAPH 11. For these problems they record the best solution time. MAC-2001$_p$ has to share the best solution time with MAC-3$_d$ and MAC-3$_p$ for GRAPH 6 and GRAPH 11. MAC-3$_d$ performs quite well. For 21 out of the 25 problems it records the best solution time and for 15 out of those 21 problems it does so for $comp$. MAC-3$_d$ solved quickly both for problems that required search and those that did not.

It should be observed that all problems have a relatively low density—it is always below 7%. It should be interesting to also compare the algorithms for larger real-world problems.

Comment 1 (The impact of counting checks) *Remember that the timings that we report on in this section do not include the time for counting checks. In a paper which was submitted for publication we reported on results from a subset of the problems that we report on in this section. However, for that paper we* did *include time for counting. It was much to our surprise when we observed that (for a $comp$ heuristic) when counting* was *included* MAC-3$_d$ *recorded a solution time of 5.345 seconds for* RLFAP 11 *whereas it required only 3.723 seconds if counting was excluded. Similarly,* MAC-3 *required 7.861 seconds with counting but 4.950 seconds without. Finally,* MAC-2001$_p$ *required 3.866 seconds with counting and 3.335 seconds without. These findings demonstrate that counting checks can have a* significant *impact on the solution time of algorithms, especially if these algorithms spend many checks. For* MAC-3*, the algorithm that requires most checks for* RLFAP 11*, the ratio of solution times is 1.588, whereas for* MAC-2001$_p$*, the algorithm that requires the fewest checks for the same problem, the ratio is only 1.159. Whereas it may be argued that including the counting of checks simulates an environment where checks are more expensive, we believe that to separate the counting and timing will result in a fairer comparison. After all,* any *algorithm that spends more than ed^2 checks is inefficient if only* (very *expensive) checks are considered, but there are many* real *applications where checks are cheap and it is only possible to find out how well an algorithm can perform on such problems by omitting the counting of checks. Also it should be interesting to find out which users in professional working environments are more interested in knowing about the number of checks carried out by the tool called* MAC *that they use to solve their daily problems and which of them are more interested in a significant reduction of its solution time.*

Comment 2 (Timing without counting) *In the light of Comment 1 it should be interesting to once more carry out our experiments with random problems this time excluding the counting of checks from the solution time. We anticipate that:*

- *The ratio between the average solution times of the* MAC-2001 *family and the lightweight family will increase;*
- *MAC-3 will become (much) more competitive compared to both* MAC-3$_p$ *and* MAC-3$_d$ *from the perspective of average solution time;*

- *MAC-3$_p$'s solution time will improve upon that of* MAC-3$_d$.

In summary, for the real-world problems that we considered MAC-2001$_p$ and MAC-2001 are the best algorithms when it comes to checks. MAC-3$_d$ recorded the quickest solution time for the vast majority of the problems. We have provided reasons why the counting of checks should be excluded from the solution time.

							Checks			Time	
Algorithm	Problem	n	e	density	ac	lex	$rlex$	$comp$	lex	$rlex$	$comp$
MAC-3	RLFAP 1	916	5548	1.3%	yes	4.241e+06	4.024e+06	4.165e+06	0.290	0.376	0.383
MAC-3_p	RLFAP 1	916	5548	1.3%	yes	3.892e+06	3.969e+06	3.635e+06	0.288	0.332	0.337
MAC-3_d	RLFAP 1	916	5548	1.3%	yes	2.598e+06	2.672e+06	1.919e+06	**0.250**	*0.298*	*0.283*
MAC-2001	RLFAP 1	916	5548	1.3%	yes	1.850e+06	1.855e+06	1.778e+06	0.285	0.391	0.378
MAC-2001_p	RLFAP 1	916	5548	1.3%	yes	*1.849e+06*	*1.852e+06*	**1.776e+06**	0.286	0.331	0.345
MAC-3	RLFAP 2	200	1235	6.2%	yes	8.678e+05	8.578e+05	8.594e+05	0.053	0.058	0.058
MAC-3_p	RLFAP 2	200	1235	6.2%	yes	8.157e+05	8.428e+05	7.386e+05	0.054	0.057	0.052
MAC-3_d	RLFAP 2	200	1235	6.2%	yes	5.680e+05	5.946e+05	**3.814e+05**	*0.047*	*0.051*	**0.042**
MAC-2001	RLFAP 2	200	1235	6.2%	yes	4.370e+05	*4.367e+05*	4.140e+05	0.056	0.064	0.060
MAC-2001_p	RLFAP 2	200	1235	6.2%	yes	*4.364e+05*	*4.367e+05*	4.132e+05	0.057	0.060	0.057
MAC-3	RLFAP 3	400	2760	3.5%	yes	2.132e+06	2.055e+06	2.101e+06	0.136	0.158	0.157
MAC-3_p	RLFAP 3	400	2760	3.5%	yes	1.978e+06	2.026e+06	1.830e+06	0.136	0.149	0.143
MAC-3_d	RLFAP 3	400	2760	3.5%	yes	1.376e+06	1.420e+06	1.006e+06	*0.121*	*0.135*	**0.120**
MAC-2001	RLFAP 3	400	2760	3.5%	yes	9.491e+05	*9.503e+05*	9.122e+05	0.139	0.168	0.159
MAC-2001_p	RLFAP 3	400	2760	3.5%	yes	*9.486e+05*	9.508e+05	**9.110e+05**	0.140	0.153	0.151
MAC-3	RLFAP 4	680	3967	1.7%	yes	2.788e+06	3.321e+06	2.271e+06	0.189	0.398	0.300
MAC-3_p	RLFAP 4	680	3967	1.7%	yes	2.406e+06	2.507e+06	1.054e+06	0.177	0.276	0.181
MAC-3_d	RLFAP 4	680	3967	1.7%	yes	1.647e+06	1.753e+06	**6.709e+05**	**0.145**	*0.249*	*0.164*
MAC-2001	RLFAP 4	680	3967	1.7%	yes	1.603e+06	1.733e+06	1.464e+06	0.169	0.369	0.298
MAC-2001_p	RLFAP 4	680	3967	1.7%	yes	*1.480e+06*	*1.490e+06*	7.628e+05	0.162	0.259	0.188

Table 6.1: Average results for real-world problems RLFAP 1–4.

Algorithm	Problem	n	e	density	ac	Checks lex	Checks $rlex$	Checks $comp$	Time lex	Time $rlex$	Time $comp$
MAC-3	RLFAP 5	400	2598	3.3%	yes	8.649e+07	2.412e+07	5.611e+06	6.723	3.196	**1.835**
MAC-3_p	RLFAP 5	400	2598	3.3%	yes	8.633e+07	2.357e+07	5.140e+06	7.217	3.417	1.885
MAC-3_d	RLFAP 5	400	2598	3.3%	yes	8.563e+07	2.307e+07	4.856e+06	7.195	3.396	1.864
MAC-2001	RLFAP 5	400	2598	3.3%	yes	1.146e+07	5.767e+06	2.458e+06	*4.298*	*2.926*	1.920
MAC-2001_p	RLFAP 5	400	2598	3.3%	yes	*1.142e+07*	*5.614e+06*	**2.192e+06**	4.451	2.966	1.914
MAC-3	RLFAP 6	200	1322	6.7%	no	9.746e+05	1.151e+06	8.044e+05	0.057	0.079	0.057
MAC-3_p	RLFAP 6	200	1322	6.7%	no	8.254e+05	9.566e+05	1.386e+05	0.051	0.066	0.015
MAC-3_d	RLFAP 6	200	1322	6.7%	no	5.610e+05	6.673e+05	**9.651e+04**	*0.040*	*0.055*	**0.014**
MAC-2001	RLFAP 6	200	1322	6.7%	no	5.706e+05	6.094e+05	5.518e+05	0.051	0.069	0.058
MAC-2001_p	RLFAP 6	200	1322	6.7%	no	*5.176e+05*	*5.721e+05*	1.191e+05	0.047	0.061	0.018
MAC-3	RLFAP 7	400	2865	3.6%	no	5.437e+05	8.121e+05	5.500e+05	0.033	0.075	0.051
MAC-3_p	RLFAP 7	400	2865	3.6%	no	4.512e+05	5.125e+05	2.888e+04	0.030	0.044	**0.004**
MAC-3_d	RLFAP 7	400	2865	3.6%	no	*2.733e+05*	*3.458e+05*	**1.684e+04**	*0.023*	*0.038*	**0.004**
MAC-2001	RLFAP 7	400	2865	3.6%	no	4.335e+05	5.178e+05	4.674e+05	0.035	0.073	0.055
MAC-2001_p	RLFAP 7	400	2865	3.6%	no	3.586e+05	3.476e+05	2.882e+04	0.032	0.043	0.005
MAC-3	RLFAP 8	916	5744	1.4%	no	8.944e+05	6.406e+05	1.171e+06	0.059	0.079	0.168
MAC-3_p	RLFAP 8	916	5744	1.4%	no	6.253e+05	6.877e+05	3.633e+04	0.045	0.078	0.016
MAC-3_d	RLFAP 8	916	5744	1.4%	no	*3.764e+05*	*4.480e+05*	**2.593e+04**	*0.035*	*0.070*	**0.015**
MAC-2001	RLFAP 8	916	5744	1.4%	no	6.887e+05	4.585e+05	9.411e+05	0.063	0.081	0.174
MAC-2001_p	RLFAP 8	916	5744	1.4%	no	4.960e+05	4.913e+05	3.383e+04	0.048	0.078	0.017

Table 6.2: Average results for real-world problems RLFAP 5–8.

Algorithm	Problem	n	e	density	ac	Checks *lex*	Checks *rlex*	Checks *comp*	Time *lex*	Time *rlex*	Time *comp*
MAC-3	RLFAP 9	680	4103	1.8%	no	4.907e+05	9.392e+05	1.064e+06	0.030	0.107	0.125
MAC-3_p	RLFAP 9	680	4103	1.8%	no	5.550e+05	6.953e+05	1.921e+05	0.038	0.074	0.034
MAC-3_d	RLFAP 9	680	4103	1.8%	no	*3.280e+05*	*4.591e+05*	**1.170e+05**	**0.029**	*0.067*	*0.030*
MAC-2001	RLFAP 9	680	4103	1.8%	no	3.917e+05	6.375e+05	8.361e+05	0.032	0.108	0.129
MAC-2001_p	RLFAP 9	680	4103	1.8%	no	4.513e+05	4.879e+05	1.725e+05	0.040	0.074	0.037
MAC-3	RLFAP 10	680	4103	1.8%	no	4.907e+05	9.392e+05	1.064e+06	0.031	0.107	0.125
MAC-3_p	RLFAP 10	680	4103	1.8%	no	5.550e+05	6.953e+05	1.921e+05	0.038	0.073	0.034
MAC-3_d	RLFAP 10	680	4103	1.8%	no	*3.280e+05*	*4.591e+05*	**1.170e+05**	**0.029**	*0.065*	*0.030*
MAC-2001	RLFAP 10	680	4103	1.8%	no	3.917e+05	6.375e+05	8.361e+05	0.032	0.108	0.129
MAC-2001_p	RLFAP 10	680	4103	1.8%	no	4.513e+05	4.879e+05	1.725e+05	0.040	0.074	0.037
MAC-3	RLFAP 11	680	4103	1.8%	yes	2.900e+08	1.566e+08	5.655e+07	19.191	12.544	4.950
MAC-3_p	RLFAP 11	680	4103	1.8%	yes	2.133e+08	1.419e+08	4.374e+07	15.767	11.887	4.152
MAC-3_d	RLFAP 11	680	4103	1.8%	yes	1.718e+08	1.154e+08	3.093e+07	14.505	11.110	3.723
MAC-2001	RLFAP 11	680	4103	1.8%	yes	3.550e+07	*2.777e+07*	1.043e+07	10.949	9.144	3.743
MAC-2001_p	RLFAP 11	680	4103	1.8%	yes	*3.484e+07*	2.906e+07	**1.039e+07**	*9.649*	*8.567*	**3.335**

Table 6.3: Average results for real-world problems RLFAP 9–11.

Algorithm	Problem	n	e	density	ac	Checks *lex*	Checks *rlex*	Checks *comp*	Time *lex*	Time *rlex*	Time *comp*
MAC-3	GRAPH 1	200	1134	5.7%	yes	8.374e+05	8.334e+05	7.794e+05	0.053	0.058	0.055
MAC-3_p	GRAPH 1	200	1134	5.7%	yes	7.969e+05	8.247e+05	7.078e+05	0.054	0.058	0.052
MAC-3_d	GRAPH 1	200	1134	5.7%	yes	5.639e+05	5.874e+05	4.103e+05	*0.046*	*0.050*	**0.041**
MAC-2001	GRAPH 1	200	1134	5.7%	yes	3.112e+05	3.119e+05	3.035e+05	0.048	0.055	0.052
MAC-2001_p	GRAPH 1	200	1134	5.7%	yes	*3.107e+05*	*3.115e+05*	**3.032e+05**	0.048	0.052	0.050
MAC-3	GRAPH 2	400	2245	2.8%	yes	1.876e+06	1.900e+06	1.705e+06	0.125	0.147	0.132
MAC-3_p	GRAPH 2	400	2245	2.8%	yes	1.828e+06	1.887e+06	1.558e+06	0.130	0.141	0.128
MAC-3_d	GRAPH 2	400	2245	2.8%	yes	1.379e+06	1.426e+06	8.789e+05	0.115	0.128	**0.104**
MAC-2001	GRAPH 2	400	2245	2.8%	yes	6.739e+05	*6.741e+05*	6.517e+05	*0.107*	0.132	0.121
MAC-2001_p	GRAPH 2	400	2245	2.8%	yes	*6.735e+05*	6.747e+05	**6.508e+05**	0.110	*0.120*	0.119
MAC-3	GRAPH 3	200	1134	5.7%	yes	1.212e+06	1.207e+06	1.103e+06	0.074	0.080	0.073
MAC-3_p	GRAPH 3	200	1134	5.7%	yes	1.195e+06	1.200e+06	1.070e+06	0.076	0.081	0.074
MAC-3_d	GRAPH 3	200	1134	5.7%	yes	9.323e+05	9.399e+05	7.906e+05	0.068	*0.072*	**0.064**
MAC-2001	GRAPH 3	200	1134	5.7%	yes	6.017e+05	*5.946e+05*	5.877e+05	*0.065*	0.073	0.069
MAC-2001_p	GRAPH 3	200	1134	5.7%	yes	*5.998e+05*	5.973e+05	**5.797e+05**	0.066	0.069	0.067
MAC-3	GRAPH 4	400	2244	2.8%	yes	2.351e+06	2.449e+06	2.168e+06	0.152	0.182	0.164
MAC-3_p	GRAPH 4	400	2244	2.8%	yes	2.331e+06	2.409e+06	2.063e+06	0.159	0.174	0.161
MAC-3_d	GRAPH 4	400	2244	2.8%	yes	1.811e+06	1.884e+06	1.521e+06	0.141	0.158	*0.139*
MAC-2001	GRAPH 4	400	2244	2.8%	yes	1.193e+06	*1.195e+06*	1.155e+06	**0.136**	0.164	0.154
MAC-2001_p	GRAPH 4	400	2244	2.8%	yes	*1.190e+06*	1.200e+06	**1.125e+06**	0.139	*0.152*	0.147
MAC-3	GRAPH 5	200	1134	5.7%	no	2.451e+05	4.323e+05	1.392e+05	0.015	0.032	0.016
MAC-3_p	GRAPH 5	200	1134	5.7%	no	2.297e+05	2.416e+05	2.614e+04	0.015	0.018	**0.004**
MAC-3_d	GRAPH 5	200	1134	5.7%	no	*1.537e+05*	1.724e+05	**1.906e+04**	*0.012*	*0.016*	**0.004**
MAC-2001	GRAPH 5	200	1134	5.7%	no	1.695e+05	2.342e+05	1.103e+05	0.015	0.029	0.018
MAC-2001_p	GRAPH 5	200	1134	5.7%	no	1.659e+05	*1.594e+05*	2.466e+04	0.017	0.018	0.005

Table 6.4: Average results for real-world problems GRAPH 1–5.

Algorithm	Problem	n	e	density	ac	Checks *lex*	Checks *rlex*	Checks *comp*	Time *lex*	Time *rlex*	Time *comp*
MAC-3	GRAPH 6	400	2170	2.7%	no	5.510e+05	7.005e+05	3.318e+05	0.035	0.064	0.039
MAC-3$_p$	GRAPH 6	400	2170	2.7%	no	5.260e+05	6.215e+05	7.957e+03	0.036	0.052	**0.003**
MAC-3$_d$	GRAPH 6	400	2170	2.7%	no	*3.325e+05*	*4.151e+05*	**5.978e+03**	*0.028*	*0.045*	**0.003**
MAC-2001	GRAPH 6	400	2170	2.7%	no	3.984e+05	4.419e+05	2.646e+05	0.035	0.062	0.042
MAC-2001$_p$	GRAPH 6	400	2170	2.7%	no	3.981e+05	4.449e+05	7.561e+03	0.037	0.052	**0.003**
MAC-3	GRAPH 7	400	2170	2.7%	no	3.852e+05	4.855e+05	3.178e+05	0.024	0.043	0.046
MAC-3$_p$	GRAPH 7	400	2170	2.7%	no	4.103e+05	4.718e+05	7.350e+03	0.028	0.042	**0.002**
MAC-3$_d$	GRAPH 7	400	2170	2.7%	no	*2.595e+05*	3.212e+05	**5.362e+03**	*0.022*	*0.037*	**0.002**
MAC-2001	GRAPH 7	400	2170	2.7%	no	2.821e+05	3.475e+05	2.471e+05	0.025	0.043	0.049
MAC-2001$_p$	GRAPH 7	400	2170	2.7%	no	3.126e+05	*3.182e+05*	6.910e+03	0.029	0.041	0.003
MAC-3	GRAPH 8	680	3757	1.6%	yes	3.141e+06	3.144e+06	2.839e+06	0.221	0.275	0.253
MAC-3$_p$	GRAPH 8	680	3757	1.6%	yes	3.040e+06	3.110e+06	2.629e+06	0.231	0.253	0.239
MAC-3$_d$	GRAPH 8	680	3757	1.6%	yes	2.194e+06	2.252e+06	1.576e+06	**0.203**	*0.230*	*0.204*
MAC-2001	GRAPH 8	680	3757	1.6%	yes	1.260e+06	*1.265e+06*	1.229e+06	0.204	0.264	0.246
MAC-2001$_p$	GRAPH 8	680	3757	1.6%	yes	*1.258e+06*	*1.265e+06*	**1.225e+06**	0.209	0.231	0.234
MAC-3	GRAPH 9	916	5246	1.3%	yes	4.425e+06	4.508e+06	3.890e+06	0.320	0.422	0.385
MAC-3$_p$	GRAPH 9	916	5246	1.3%	yes	4.332e+06	4.474e+06	3.586e+06	0.338	0.380	0.357
MAC-3$_d$	GRAPH 9	916	5246	1.3%	yes	3.305e+06	3.434e+06	2.171e+06	**0.308**	*0.353*	*0.311*
MAC-2001	GRAPH 9	916	5246	1.3%	yes	1.867e+06	*1.870e+06*	1.796e+06	0.311	0.416	0.391
MAC-2001$_p$	GRAPH 9	916	5246	1.3%	yes	*1.865e+06*	1.871e+06	**1.792e+06**	0.320	0.358	0.366
MAC-3	GRAPH 10	680	3907	1.7%	yes	8.253e+06	8.302e+06	5.678e+06	0.583	0.713	0.562
MAC-3$_p$	GRAPH 10	680	3907	1.7%	yes	8.075e+06	8.573e+06	5.501e+06	0.601	0.725	0.544
MAC-3$_d$	GRAPH 10	680	3907	1.7%	yes	7.019e+06	7.488e+06	4.294e+06	0.567	0.693	0.499
MAC-2001	GRAPH 10	680	3907	1.7%	yes	2.689e+06	*2.681e+06*	2.354e+06	**0.451**	0.588	0.513
MAC-2001$_p$	GRAPH 10	680	3907	1.7%	yes	*2.672e+06*	2.739e+06	**2.332e+06**	0.456	*0.573*	*0.484*

Table 6.5: Average results for real-world problems GRAPH 6–10.

Algorithm	Problem	n	e	density	ac	Checks *lex*	Checks *rlex*	Checks *comp*	Time *lex*	Time *rlex*	Time *comp*
MAC-3	GRAPH 11	680	3757	1.6%	no	2.246e+05	2.886e+05	6.563e+05	*0.015*	*0.031*	0.127
MAC-3_p	GRAPH 11	680	3757	1.6%	no	3.024e+05	3.512e+05	8.966e+03	0.022	0.036	**0.004**
MAC-3_d	GRAPH 11	680	3757	1.6%	no	*1.779e+05*	*2.270e+05*	**6.886e+03**	0.017	0.033	**0.004**
MAC-2001	GRAPH 11	680	3757	1.6%	no	1.967e+05	2.344e+05	5.069e+05	0.017	0.033	0.133
MAC-2001_p	GRAPH 11	680	3757	1.6%	no	2.679e+05	2.784e+05	8.470e+03	0.025	0.038	**0.004**
MAC-3	GRAPH 12	680	4017	1.7%	no	3.240e+05	6.899e+05	4.535e+05	*0.022*	0.070	0.074
MAC-3_p	GRAPH 12	680	4017	1.7%	no	4.126e+05	4.806e+05	6.737e+04	0.030	0.050	0.020
MAC-3_d	GRAPH 12	680	4017	1.7%	no	*2.504e+05*	*3.286e+05*	**4.912e+04**	0.023	*0.046*	**0.019**
MAC-2001	GRAPH 12	680	4017	1.7%	no	2.796e+05	5.406e+05	3.964e+05	0.024	0.073	0.079
MAC-2001_p	GRAPH 12	680	4017	1.7%	no	3.483e+05	3.546e+05	6.405e+04	0.033	0.051	0.023
MAC-3	GRAPH 13	916	5273	1.3%	no	6.151e+05	6.737e+05	4.737e+05	*0.040*	*0.072*	0.095
MAC-3_p	GRAPH 13	916	5273	1.3%	no	7.480e+05	8.668e+05	2.378e+04	0.052	0.090	**0.010**
MAC-3_d	GRAPH 13	916	5273	1.3%	no	*4.552e+05*	5.753e+05	**1.776e+04**	0.041	0.082	**0.010**
MAC-2001	GRAPH 13	916	5273	1.3%	no	4.890e+05	*5.729e+05*	4.126e+05	0.042	0.077	0.102
MAC-2001_p	GRAPH 13	916	5273	1.3%	no	6.234e+05	6.550e+05	2.244e+04	0.056	0.092	0.011
MAC-3	GRAPH 14	916	4638	1.1%	yes	3.894e+06	3.945e+06	3.401e+06	0.282	0.364	0.330
MAC-3_p	GRAPH 14	916	4638	1.1%	yes	3.827e+06	3.923e+06	3.090e+06	0.299	0.332	0.308
MAC-3_d	GRAPH 14	916	4638	1.1%	yes	2.866e+06	2.957e+06	1.734e+06	0.267	*0.304*	**0.259**
MAC-2001	GRAPH 14	916	4638	1.1%	yes	*1.651e+06*	*1.652e+06*	1.591e+06	*0.263*	0.351	0.327
MAC-2001_p	GRAPH 14	916	4638	1.1%	yes	*1.651e+06*	*1.652e+06*	**1.589e+06**	0.271	*0.304*	0.311

Table 6.6: Average results for real-world problems GRAPH 11–14.

Chapter 7

Conclusions and Recommendations

We compared five algorithms called MAC-2001, MAC-2001_p, MAC-3, MAC-3_p, and MAC-3_d. MAC-2001 and MAC-2001_p have an arc-consistency component with an optimal worst case time-complexity. The remaining algorithms do not. We demonstrated that MAC-2001's space-complexity is $\mathcal{O}(ed \min(n, d))$ and we demonstrated that this size may be prohibitive even for CSPs that are relatively easy. We compared the algorithms for search and for three different arc-heuristics, called lex, $rlex$, and $comp$. We considered random problems where checks are cheap and real-world problems where checks are expensive. For the random problems we included the counting of checks in the solution time. For the real-world problems we did not. For the random problems our findings are that good arc-consistency algorithms do not always need to have an optimal worst case time-complexity. We presented results that suggest quite the opposite. For a given arc-heuristic MAC-2001 and MAC-2001_p always required more solution time than the others. MAC-3_d with $comp$ arc-heuristic, was the most efficient combination when it comes to saving time. MAC-2001_p required about 21% more time on average than MAC-3_d and MAC-2001 required about 34% more. For the real-world problems things were not as clear. For these problems MAC-2001 and MAC-2001_p were the best in saving checks but MAC-3_d with a $comp$ arc-heuristic recorded the best solution time for the vast majority of these problems. Since the differences were not as clear as for the random problems it is difficult to say which algorithm should be preferred for the real-world problems that we considered in our test. Finally, we have observed that including the counting of checks in the the solution time results in an increase of about 59% for some algorithms. It is for this reason that we argue that to find out how well algorithms perform (at least where problems are easy or where checks are cheap) the counting of checks should be separated from measuring the solution time. We anticipate that if we were to compare the algorithms once more for random problems, this time separating the counting and timing, then the ratio between the best solution time from the MAC-2001 family and the best solution time from the lightweight algorithms will increase. To conduct such experiment is something we intend to do in the near future.

Acknowledgements

First of all I should like to thank Christian Bessière for letting us use his solver in an early stage of this work and for useful discussions. I should like to thank Rick Wallace and Ken Brown for early discussions. Also I wish to thank Christian van den Bosch for setting up and carrying out the experiments. Finally, I should like to express my gratitude to Gene Freuder for his support of this work. This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

Bibliography

[Bessière and Régin, 2001] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 309–315, 2001.

[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*, 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.

[CELAR, 1994] CELAR. Radio link frequency assignment problem benchmark, `ftp://ftp.cs.city.ac.uk/pub/constraints/csp-benchmarks/celar`, 1994.

[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.*, 1996] I.P. Gent, MacIntyre E., P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In E.C. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 179–193. Springer, 1996.

[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*, pages 327–340. Springer, 1997.

[Gent *et al.*, 2001] Ian Gent, Ewan MacIntyre, Patrick Prosser, Barbara Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.

[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.

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

[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*, pages 125–129. John Wiley and 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*, pages 140–149, 2000.

[van Dongen, 2002a] M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture notes in Computer Science*, pages 755–760. Springer, 2002.

[van Dongen, 2002b] M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. Technical Report TR-01-2002, Cork Constraint Computation Centre, 2002.

[van Dongen, 2003a] M.R.C. van Dongen. Domain-heuristics for arc-consistency algorithms. In B. O'Sullivan, editor, *Recent Advances in Constraints*, volume 2627 of *Lecture Notes in Artificial Intelligence*, pages 61–75. Springer, 2003. To be published.

[van Dongen, 2003b] M.R.C. van Dongen. Lightweight arc-consistency algorithms. Technical Report TR-01-2003, Cork Constraint Computation Centre, 2003.

[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.

[Zhang and Yap, 2001] Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 316–321, 2001.