To Avoid Repeating Checks Does Not Always Save Time

M.R.C. van Dongen (`dongen@cs.ucc.ie`)[1]
Cork Constraint Computation Centre/Computer Science Department
University College Cork

Abstract. Arc-consistency algorithms are the workhorse of backtrackers that Maintain Arc-Consistency (MAC). This paper 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 was faster for 40% of the real-world problems but slower for the remaining real-world problems. Our results are an indication that if checks are cheap then lightweight algorithms like MAC-3_d are promising.

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 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; 2002]. 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. Finally, we shall introduce some notation for compactly describing ordering heuristics.

For random and real-world problems and for as far as avoiding the re-discovery of checks is concerned 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 things were not as clear.

The results presented in this paper 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 paper is organised as follows. Section 2 is an introduction to constraint satisfaction. Section 3 presents some notation for describing selection heuristics. Section 4 describes 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 experimental results. Conclusions are presented in Section 7.

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 $(\, X, D, C \,)$,

[1] This work has received support from Science Foundation Ireland under Grant 00/PI.1/C075.

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 $k(v) = | D(v) |$, 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.

3 Operators for Composing Selection Heuristics

In this section 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 section and return to it later. 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:

$$v \preceq_2 \bullet \preceq_1 w \iff \begin{aligned}&(v \preceq_1 w \wedge \neg w \preceq_1 v) \vee \\ &(v \preceq_1 w \wedge w \preceq_1 v \wedge v \preceq_2 w)\,.\end{aligned}$$

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 $\otimes^{\#}_{\leq} \bullet \otimes^{k}_{\leq}$, 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 $\otimes^{\#}_{\leq} \bullet \otimes^{\delta_c}_{\geq} \bullet \otimes^{k}_{\leq}$, 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.

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. 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.

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 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, 2002]. 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 Algorithm $\mathcal{D}$ from [van Dongen, 2001]. 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.

5 Description of Algorithms

In this section 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. We shall also prove that MAC-2001 has a $\mathcal{O}(ed \min(n, d))$ space-complexity.

5.1 MAC-3_d and MAC-3_p

AC-3_d is a cross-breed between AC-3 and DEE [Mackworth, 1977; Gaschnig, 1978]. The only difference between AC-3 and AC-3_d is that if AC-3_d's arc-heuristic select the arc (x, y) from the queue and if the reverse arc (y, x) is also in the queue then AC-3_d will remove both arcs from the queue and and will simultaneously revise *two* domains with algorithm $\mathcal{D}$ described in [van Dongen, 2001; 2003a]. $\mathcal{D}$ uses a *double-support* domain-heuristic, i.e. a heuristic which prefers double-support checks. 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 its call to $\mathcal{D}$ by two calls to Mackworth's *revise* to *sequentially* revise two domains with one constraint. The difference between AC-3_p and AC-3_d is AC-3_d's double-support heuristic. 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 $\mathcal{O}(e + nd)$.

5.2 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 1 and 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. 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 $D(y)$.

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.

```
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 1. 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 2. *revise*-2001.

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.

AC-2001$_p$ 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.

6 Experimental Results

In this section we shall 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.

6.1 Implementation Details

All implementations used the same basic data structures as used by MAC-3$_d$. The implementations of MAC-2001 and MAC-2001$_p$ 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 and it 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 $f(v) = k(v)/\delta_o(v)$. We considered three different arc-heuristics, called lex, $rlex$, and $comp$. Using the notation introduced in Section 3 these can be defined as:

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

At the moment of writing $comp$ is the best known arc-heuristic for MAC-3$_d$. Further in this section 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 on 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.

6.2 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 total time for our comparison is equivalent to more than 100 days of computation on a single machine.

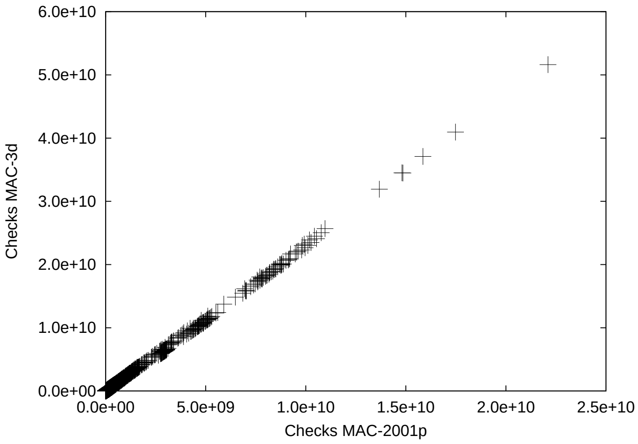

Figure 3. Scatter plot of checks for MAC-3$_d$ and *comp* vs. checks for MAC-2001$_p$ and *comp* for search and problem size 30.

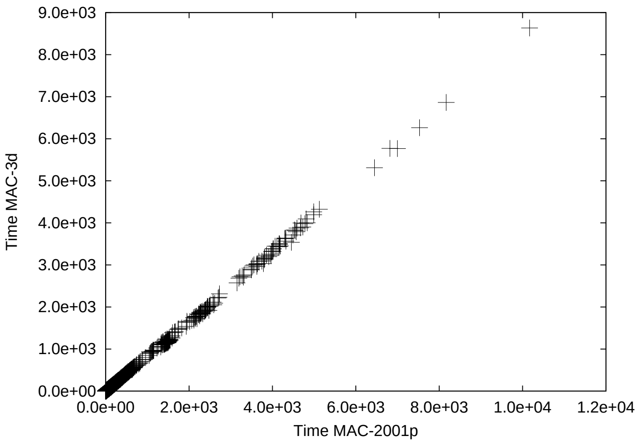

Figure 4. Scatter plot of time for MAC-3$_d$ and *comp* vs. time for MAC-2001$_p$ and *comp* for search and problem size 30.

For random problems, the best lightweight algorithm turned out to be MAC-3$_d$ with a *comp* heuristic. The best algorithm from the MAC-2001 family was MAC-2001$_p$ with a *comp* heuristic. Figure 3 depicts a scatter plot of the checks required by MAC-3$_d$ with *comp* versus the number of checks required by MAC-2001$_p$ with *comp* for problem size 30. Figure 4 depicts a scatter plot of the time required by MAC-3$_d$ with *comp* versus the time required by MAC-2001$_p$ with *comp* for problem size 30. Both figures suggest that there is a linear relationship between the number of checks required by MAC-3_d and MAC-2001_p and between the solution times of MAC-3_d and MAC-2001_p. Similar linear relationships were observed for other combinations of algorithms.

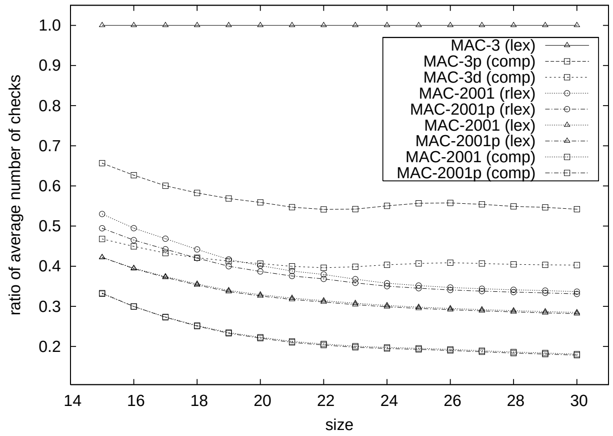

Figure 5. Ratio of average #checks vs. problem size for random problems and search. For each size the average #checks is divided by the average #checks required by MAC-3 with lex.

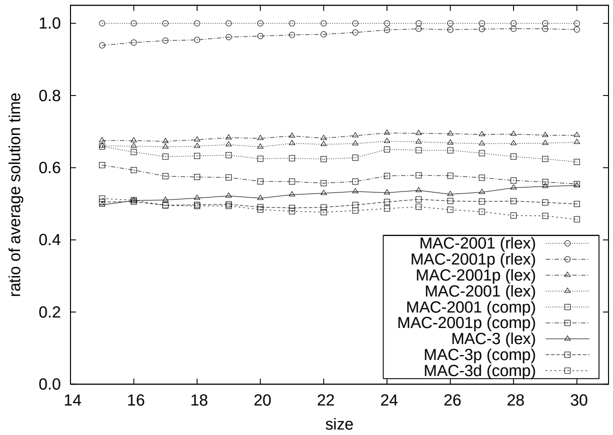

Figure 6. Ratio of average solution time vs. problem size for random problems and search. For each size the average time is divided by the average time required by MAC-2001 with $rlex$.

Figure 5 depicts the ratio between the average number of checks on the one hand and the average number required by 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 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 5 are two pairs of lines. The pair at the bottom corresponds to MAC-2001 and MAC-2001_p with a $comp$ heuristic. The other pair corresponds to MAC-2001 and MAC-2001_p with a lex heuristic. As the problem size increases the lines for MAC-2001 and MAC-2001_p with an $rlex$ heuristic also seem to converge. MAC-2001_p and MAC-2001 with a $comp$ heuristic are the best when it comes to saving checks.

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 and for 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, 2002; 2003b]. The results about MAC-2001 and MAC-3 are in contrast with other results from the literature [Bessière and Régin, 2001]. However, this should not be a reason for dismissing these findings; 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 its strategy for propagation does not seem to hinder MAC-3 at all when checks are cheap.

Figures 5 and 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. Investigation of the test data revealed that this was 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.

6.3 Real-World Problems

The real-world problems came from the CELAR suite [CELAR, 1994]. We did not consider optimisation but only considered satisfiability. The same problems were considered as in [Bessière and Régin, 2001]. These problems have become a sort of a standard benchmark for real-world problems. However, see our comments further on about these problems. For every problem we computed the average solution time over 50 runs. Checks were implemented as function calls and were more expensive than for random problems. For all problems the domain size was equal to 44.

The results for the tests are depicted in Table 1. Due to space-restrictions the results for MAC-2001 have been omitted. MAC-2001 performed about the same as MAC-2001_p but required more checks and time on average. For each problem the least average number of checks and the least average solution time for that problem for all arc-heuristics are printed in bold face. For each of the remaining heuristics the least average number of checks and least average solution time are printed italicised. Again MAC-2001_p is the best algorithm when it comes to saving checks. This time it pays off. MAC-2001_p also seems to be the best algorithm when it comes to solving quickly. MAC-3_d records the least solution time for RLFAP 1, GRAPH 9 and GRAPH 14. These are the problems for which $ed \min(n, d)$ has the first and third largest size. The larger the problems become, the better MAC-3_d starts to perform relative to the performance of MAC-2001 and MAC-2001_p. It is only for the smaller problems that MAC-2001 and MAC-2001_p are the best. A possible critique is that the density of these problems is rather low—it is always below 2%. It should be interesting to compare the algorithms for larger real-world problems.

For the real-world problems that we considered MAC-2001_p and MAC-2001 are the best algorithms both in time and checks. The sparsity of the constraint graph of these problems suits both MAC-2001 and MAC-2001_p very well. There does not seem to be much between them and MAC-3_d or MAC-3_p with a $comp$ heuristic.

Algorithm	Problem	n	e	Checks lex	Checks $rlex$	Checks $comp$	Time lex	Time $rlex$	Time $comp$
MAC-3	RLFAP 1	916	5548	4.24e+06	4.02e+06	4.17e+06	0.50	0.58	0.59
MAC-3_p	RLFAP 1	916	5548	3.89e+06	3.97e+06	3.64e+06	0.47	0.52	0.51
MAC-3_d	RLFAP 1	916	5548	2.60e+06	2.67e+06	1.92e+06	**0.38**	0.43	**0.38**
MAC-2001_p	RLFAP 1	916	5548	*1.85e+06*	*1.85e+06*	**1.78e+06**	**0.38**	*0.43*	0.44
MAC-3	RLFAP 11	680	4103	2.90e+08	1.57e+08	5.66e+07	34.12	20.63	7.86
MAC-3_p	RLFAP 11	680	4103	2.13e+08	1.42e+08	4.37e+07	25.78	18.60	6.20
MAC-3_d	RLFAP 11	680	4103	1.72e+08	1.15e+08	3.09e+07	23.10	16.91	5.35
MAC-2001_p	RLFAP 11	680	4103	*3.48e+07*	2.91e+07	**1.04e+07**	*11.74*	*10.25*	**3.87**
MAC-3	GRAPH 9	916	5246	4.43e+06	4.51e+06	3.90e+06	0.54	0.65	0.58
MAC-3_p	GRAPH 9	916	5246	4.33e+06	4.48e+06	3.59e+06	0.54	0.60	0.52
MAC-3_d	GRAPH 9	916	5246	3.31e+06	3.43e+06	2.18e+06	0.47	0.52	**0.42**
MAC-2001_p	GRAPH 9	916	5246	*1.86e+06*	1.87e+06	**1.79e+06**	**0.42**	*0.46*	0.46
MAC-3	GRAPH 10	680	3907	8.25e+06	8.30e+06	5.68e+06	1.00	1.13	0.85
MAC-3_p	GRAPH 10	680	3907	8.08e+06	8.57e+06	5.50e+06	0.98	1.13	0.80
MAC-3_d	GRAPH 10	680	3907	7.02e+06	7.49e+06	4.29e+06	0.90	1.05	0.71
MAC-2001_p	GRAPH 10	680	3907	*2.67e+06*	2.74e+06	**2.33e+06**	**0.60**	*0.72*	*0.61*
MAC-3	GRAPH 14	916	4638	3.89e+06	3.95e+06	3.40e+06	0.48	0.56	0.50
MAC-3_p	GRAPH 14	916	4638	3.83e+06	3.92e+06	3.09e+06	0.48	0.51	0.45
MAC-3_d	GRAPH 14	916	4638	2.87e+06	2.96e+06	1.73e+06	0.41	0.45	**0.34**
MAC-2001_p	GRAPH 14	916	4638	*1.65e+06*	*1.65e+06*	**1.59e+06**	*0.36*	*0.39*	0.39

Table 1. Average results for real-world problems.

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 easy CSPs. 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 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 and a $comp$ 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. Here MAC-2001 and MAC-2001_p were the best in solving quickly but MAC-3_d with a $comp$ arc-heuristic was not much worse.

REFERENCES

[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.

[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 2^{nd} 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 3^{rd} 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. *Journal of 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 11^{th} European Conference on Artificial Intelligence*, pages 125–129. John Wiley and Sons, 1994.

[van Dongen, 2001] M.R.C. van Dongen. Domain heuristics for arc-consistency algorithms. In *Proceedings of the 12^{th} Irish Conference on Artificial Intelligence and Cognitive Science*, pages 179–188, 2001.

[van Dongen, 2002] 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 8^{th} International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture notes in Computer Science*, pages 755–760. Springer, 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.

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