Two New Lightweight Arc Consistency Algorithms

D. Mehta and M.R.C. van Dongen*

Boole Centre for Research in Informatics/Cork Constraint Computation Centre
Computer Science Department, University College Cork

Abstract. Coarse-grained arc consistency algorithms like AC-3, AC-3_d, and AC-2001, are efficient when it comes to transforming a Constraint Satisfaction Problem (CSP) to its arc consistent equivalent. These algorithms repeatedly carry out revisions to remove unsupported values from the domains of the variables. The order of these revisions is determined by so-called *revision ordering heuristics*. In this paper, we classify revision ordering heuristics into three different categories: *arc based*, *variable based*, and *reverse variable based* revision ordering heuristics. We point out advantages of using reverse variable based revision ordering heuristics and propose two new lightweight arc consistency algorithms AC-3_{dl} and AC-3_{ds}, which exploit these advantages. Both algorithms are equipped with domain heuristics which are inspired by AC-3_d's *double support heuristic*. AC-3_{dl} uses a lazy version of a double support heuristic while AC-3_{ds} uses AC-3_d's double support heuristic with a minor change. We experimentally compare MAC-3, MAC-3_d, MAC-3_{dl} and MAC-3_{ds}. MAC-3_{ds} is the best in saving checks. MAC-3_{dl} is good in saving time and checks on average. Experimental results demonstrate that lightweight algorithms based on reverse variable based revision ordering heuristics are good in saving checks as well as time.

1 Introduction

Arc consistency algorithms are widely used to prune the search space of Constraint Satisfaction Problems (CSPs). They use support checks to reduce the search space of CSPs. Many arc consistency algorithms have been proposed. On one side there are heavyweight arc consistency algorithms such as AC-4 [11], AC-2001 [3], AC-3.1 [18], AC-6 [1], and AC-7 [2] that use additional data structures to avoid repeating their support checks. All these algorithms have an optimal worst case time complexity of $\mathcal{O}(e\,d^2)$ where e is the number of constraints and d is the maximum domain size of the variables. On the other side there are lightweight arc consistency algorithms such as AC-3 [8], AC-3_d [13], and AC-3_p [15] which do not use additional data structures. These algorithms repeat their support checks and have a non-optimal bound of $\mathcal{O}(e\,d^3)$ for their worst case time complexity. However, despite the fact that these algorithms do not have an optimal worst case time complexity, experimental evaluation of these algorithms has demonstrated that they are efficient on average [15, 16].

Since the introduction of AC-4, most research in arc consistency algorithms is to avoid repeating checks by using additional data structures. The belief is that reducing checks helps solving problems more quickly. However, there are many instances where

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

checks are cheap and allowing algorithms to repeat their checks relieves them from the burden of maintaining a large additional bookkeeping and this may save time [15, 16]. Reducing support checks *alone* does not always help in reducing the solution time. Queue maintenance, revision ordering heuristics and domain heuristics also play an important part in reducing the time for problem solving.

In this paper, we classify revision ordering heuristics into *arc based*, *variable based*, and *reverse variable based* revision ordering heuristics. Algorithms based on reverse variable based revision ordering heuristics are good in saving checks as well as time. We only consider lightweight arc consistency algorithms. We propose two new competitive lightweight arc consistency algorithms AC-3_{dl} and AC-3_{ds} which use reverse variable based revision ordering heuristics for selecting a collection of arcs that determine the revision of the domain of the same variable. AC-3_{dl} uses a *lazy* version of a double support heuristic [13] while AC-3_{ds} uses a *strong* double support heuristic as used by AC-3_d with a small change. For real world problems MAC-3_{dl} becomes the quickest solver on average. For random problems MAC-3 equipped with variable based heuristic is the fastest. MAC-3_{ds} is the best in saving checks.

The remainder of paper is organised as follows. Section 2 is a brief introduction to constraints. Section 3 discusses related work, classifies revision ordering heuristics and presents a reverse variable based version of AC-3. Section 4 describes the new arc consistency algorithms. Section 5 presents experimental results. Conclusions are presented in Section 6.

2 Constraint Satisfaction

A *Constraint Satisfaction Problem* is defined as a set V of n variables, a non-empty domain $D(v)$ for each variable $v \in V$ and a set of e constraints among subsets of variables of V. A binary constraint C_{vw} between variables v and w is a subset of the Cartesian product of $D(v)$ and $D(w)$ that specifies the allowed pairs of values for v and w. We only consider CSPs whose constraints are binary. With each binary constraint between variables v and w we associate two arcs (v, w) and (w, v). We call v the *first variable* of the arc (v, w) and w the *second variable* of the arc (v, w).

A value $y \in D(w)$ is called a *support* for $x \in D(v)$ if the pair $(x, y) \in C_{vw}$. Similarly $x \in D(v)$ is called a support for $y \in D(w)$ if the pair $(x, y) \in C_{vw}$. A *support check* (consistency check) is a test to find if two values support each other. A value $x \in D(v)$ is *viable* if for every variable w such that C_{vw} exists x is supported by at least one value in $D(w)$. A CSP is called *arc consistent* if for every variable $v \in V$, each value $x \in D(v)$ is viable.

The *tightness* of the constraint C_{vw} is defined as $1 - | C_{vw} |/| D(v) \times D(w) |$. The *density* of a CSP is defined as $2\,e/(n^2 - n)$. The *degree* of a variable is the number of constraints involving that variable. MAC [12] is a backtrack algorithm that maintains arc consistency during search. MAC-*X* uses AC-*X* for maintaining arc consistency during search.

3 Related Literature

3.1 Introduction

As mentioned in Section 1, we only consider lightweight arc consistency algorithms. Coarse-grained lightweight arc consistency algorithms such as AC-3 [8] and AC-3_d [13] have been developed, the principle of which is to apply successive *revisions* of the domains of the variables until the problem is made arc consistent or the domain of a variable is wiped out. Here a revision of the domain of v using the constraint between v and w means to remove the values from $D(v)$ that are not supported by w. AC-3 has a $\mathcal{O}(e\,d^3)$ bound for its worst case time complexity [9] and a $\mathcal{O}(e + n\,d)$ space complexity. AC-3 cannot remember all of its support checks.

Lightweight arc consistency algorithms such as AC-3 and AC-3_d use revision ordering heuristics to select an arc (v, w) for the next revision. The arc (v, w) represents the fact that $D(v)$ will be revised against $D(w)$. A revision is called *effective* if it results in a reduction of the domain of v. Besides revision ordering heuristics there are also domain heuristics. Given the arc determining the next revision, they determine the values to be used for the next support check.

AC-3_d [13] is a cross-breed between AC-3 [8] and DEE [5]. AC-3_d's revision ordering heuristic selects an arc (v, w) for the next revision. If the arc (w, v) is also present in the queue then AC-3_d simultaneously revises the domains of v and w using the double support domain heuristic $\mathcal{D}$ described in [13]. If (w, v) is not present in the queue then it proceeds like AC-3 by using Mackworth's *revise* to relax v against w.

The double support domain heuristic $\mathcal{D}$ prefers checks between two values whose support statuses are both unknown. In $\mathcal{D}$ the row support are the values in $D(v)$ that are supported by w and the column support are the values in $D(w)$ that are supported by v. If AC-3_d simultaneously revises two domains then after computing the row support, column support is computed only for those values of w which have not provided support for values in $D(v)$ while computing the row support. AC-3_d inherits its time complexity and space complexity from AC-3.

3.2 Revision Ordering Heuristics

Revision ordering heuristics determine the next revision. Wallace and Freuder [17] pointed out that these heuristics can influence the efficiency of arc consistency algorithms. They can be classified into three categories: *arc based*, *variable based*, and *reverse variable based* revision ordering heuristics. The differences between arc based, variable based, and reverse variable based heuristics are as follows.

Arc based revision ordering heuristics are the most commonly presented. Given some selection criterion they select an arc (v, w) for the next revision. For this class of heuristics candidate arcs are stored in a data structure called a *queue*, which usually corresponds to a set or a list. Selecting the best arc from the queue can be expensive because of the following reasons. Selecting the best arc from a set or a list based queue requires $\mathcal{O}(e)$ time. However, [16] describes selection criteria for which the queue can be represented by more efficient data structures facilitating a more efficient $\mathcal{O}(n)$ selection. Second, each selected arc corresponds to exactly one revision. Since there may be many revisions there may be many selections. However in some cases like in AC-3$_d$ or in AC-3$_p$ when the reverse arc is present in the queue it allows for two simultaneous revisions. Arc based revision ordering heuristics update a queue after every effective revision, and many updates can be an overhead too.

Variable based heuristics [10] always select a variable v and repeatedly use arcs of the form (w, v) for revision until no more such arcs exist or some $D(w)$ becomes empty. Variable based heuristics may be regarded as *propagation based heuristics* because they propagate the consequences of the removal of one or more values from the domain of v. For the most commonly occurring variable based heuristics the time complexity of picking the most promising candidate v is only $\mathcal{O}(n)$ and for these heuristics the queue can be implemented as a set of variables. If variable v is selected the domain of all its neighbours in the constraint graph will be revised against the domain of v. In this setting the number of selections from the queue is usually less than the number of selections of arcs from an arc based queue. In general more checks will be required but time can be saved because the queue needs fewer selections. Here too every effective revision results in updating the queue.

Reverse variable based heuristics always select a variable v and repeatedly revise using arcs of the form (v, w) until there are no more such arcs or $D(v)$ becomes empty. Reverse variable based heuristics may be regarded as *support based heuristics* because for one variable v at a time, they seek support for each value in $D(v)$ with respect to all of its neighbours for which it is currently unknown whether such support exists. It was shown in [16] that for certain classes of heuristics a proper representation for the queue facilitates $\mathcal{O}(n)$ selection for the most promising variable v. Using this representation, the overhead of selecting the next arc (v, w) for revision is $\mathcal{O}(1)$ or $\mathcal{O}(n)$ depending upon the criterion for selecting w. When a variable v is selected a number of revisions is performed which is between 1 and the number of arcs of the form (v, w) currently present in the queue. Therefore, the number of selections (of v), and the overhead of queue management, is usually less than for arc based heuristics. Unlike arc based and variable based heuristics, the queue is less likely to be updated after every effective revision as will be shown further in this section and empirically in Section 5.

We shall use the notation proposed in [15] for describing and composing heuristics for selecting variables and arcs. Let $\delta_o(v)$ be the original degree of v, let $\delta_c(v)$ be the current degree of v, let $\#(v)$ be a unique number for v, let $s(v)$ be the current domain size of v, and let $\delta_{cn}(v)$ be the number of current neighbours w of v such that (v, w) currently present in the queue. Finally, let $\pi_i((v_1, \ldots, v_n)) = v_i$ denote the *i-th projection operator*. The *composition* of order $\preceq_2$ and linear quasi-order $\preceq_1$ is denoted by $\preceq_2 \bullet \preceq_1$. Selection is done using $\preceq_1$ and ties are broken using $\preceq_2$. Composition associates to the left. The result of *lifting* linear quasi-order $\preceq$ and function f is denoted $\otimes^f_{\preceq}$. It is the linear quasi-order such that $v \otimes^f_{\preceq} w$ if and only if $f(v) \preceq f(w)$. For example, using this notation the dom/deg variable ordering heuristic with a lexicographical tie breaker can be described as $\otimes^{\#}_{\leq} \bullet \otimes^{f}_{\leq}$, where $f(v) = s(v)/\delta_c(v)$. The lexicographical arc selection heuristic can be described as $\otimes^{\# \circ \pi_2}_{\leq} \bullet \otimes^{\# \circ \pi_1}_{\geq}$. The reader is referred to [15] for more examples and further details.

3.3 AC-3 with Reverse Variable based Revision Ordering Heuristics

Selecting a variable v and relaxing it against all of its neighbours w such that (v, w) is currently present in the queue we call a *complete relaxation* of v. Complete relaxation can be achieved efficiently using reverse variable based heuristics. Pseudo-code for a reverse variable based implementation of AC-3 is depicted in Figure 1. Pseudo-code for the function $revise$ [8] upon which AC-3 relies is depicted in Figure 2.

```
function AC-3: Boolean;
begin
  Q := { (v, w) ∈ G : v and w are neighbours };
  while Q not empty do begin
     select any v from { v : (v, w) ∈ Q };
     effective_revisions := 0;
     for each w such that (v, w) ∈ Q do begin
        remove (v, w) from Q;
        revise(v, w, change_v);
        if D(v) = ∅ then
          return False;
        else if change_v then
          effective_revisions := effective_revisions + 1;
          u := w;
        fi;
     end;
     if effective_revisions = 1 then
        Q := Q ∪ { (w', v) : w' ≠ u, w' is a neighbour of v } ;
     else if effective_revisions > 1 then
        Q := Q ∪ { (w', v) : w' is a neighbour of v };
     fi;
  end;
  return True;
end;
```

Fig. 1. The AC-3 version with reverse variable based revision ordering heuristics

It is argued in [17] that if a value is deleted from $D(v)$ when v is relaxed against w then less work needs to be done if other arcs that involve v as a second variable are revised after the revision of (v, w). We argue that even less work in terms of support checks needs to be done if other arcs that involve v as a second variable are revised after *completely* relaxing v. It will be shown further in this paper that certain classes of reverse variable based revision ordering heuristics allow the saving of checks when compared to the best known arc and variable based heuristics.

In Figure 1, if $D(v)$ was changed after completely relaxing $D(v)$ and if this was the result of *only one* effective revision ($effective\_revisions = 1$), which happened to be against $D(u)$, then all the arcs $\{ (w', v) \in G \}$ are added to the queue, *except for* (u, v), where G is the constraint graph. However, if $D(v)$ was changed as the result of *more than one* effective revision ($effective\_revisions > 1$) then *all* the arcs $\{ (w', v) \in G \}$ are added to the queue. Modulo constraint propagation effects this saves work for maintaining the queue compared to the original AC-3.

```
function revise(v, w, var change_v): Boolean;
begin
  change_v := False ;
  for each r ∈ D(v) do begin
     if ∄c ∈ D(w) such that c supports r then
       D(v) := D(v)\{r} ;
       change_v := True ;
     fi;
  return D(v) ≠ ∅;
end;
```

Fig. 2. Algorithm *revise*

4 Description of the AC-3$_{dl}$ and AC-3$_{ds}$ Algorithms

4.1 Introduction

We describe two new lightweight arc consistency algorithms which are inspired by reverse variable based heuristics and AC-3$_d$'s double support heuristic [13]. Here, a double support heuristic prefers checks between two values each of whose support statuses are unknown. It is recalled that a reverse variable based heuristic selects a variable v and repeatedly select an arc of the form (v, w) for the next revision until no more such arc exists. If during this process $D(v)$ becomes empty then the process is aborted.

```
function AC-3_d*: Boolean;
begin
  Q := { (v, w) ∈ G : v and w are neighbours };
  while Q ≠ ∅ do begin
     select any v from {v : (v, w) ∈ Q };
     effective_revisions := 0;
     \* compute row support *\
     for each w such that (v, w) ∈ Q do begin
        remove (v, w) from Q;
        compute_row_support(v, w, change_v);
        if D(v) = ∅ then
          return False;
        else if change_v then
          effective_revisions := effective_revisions + 1;
          u := w;
        fi;
     end;
     if effective_revisions = 1 then
        Q := Q ∪ { (w', v) : w' ≠ u, w' is a neighbour of v } ;
     else if effective_revisions > 1 then
        Q := Q ∪ { (w', v) : w' is a neighbour of v };
     fi;
     \* compute column support *\
     for each w such that row-support of (v, w) is computed and (w, v) ∈ Q do begin
        remove (w, v) from Q;
        compute_column_support(w, v, change_w);
        if change_w then
          Q := Q ∪ {(v', w) : v' ≠ v , v' is a neighbour of w };
        fi;
     end;
  end;
  return True;
end;
```

Fig. 3. AC-3$_{dl}$ / AC-3$_{ds}$ Algorithm

Pseudo-code for AC-3_{dl} and AC-3_{ds} is depicted in Figure 3. The difference between them is their domain heuristic: the way they compute the *row support* and *the column support*. For each individual arc (v, w) the *row support* are the values in $D(v)$ that are supported by the values in $D(w)$ and the *column support* are the values in $D(w)$ that are supported by the values in $D(v)$. AC-3_{dl} uses a lazy version of AC-3_d's double support heuristic while AC-3_{ds} uses AC-3_d's double support heuristic with a small change. AC-3_{dl}'s (AC-3_{ds}'s) algorithm for computing row support and column support are depicted in Figures 4 and 5 (Figures 6 and 7).

After selecting a variable v the procedure as shown in Figure 3 is divided into two phases. The first phase computes the row support for all arcs of the form (v, w) that are in the queue. The second phase computes the column support for the arcs (v, w) whose row support has just been computed and for which the reverse arc (w, v) is also in the queue. Only row support computations can lead to a wipe out. This is why column support computations are postponed: they cannot result in wipeouts.

The main difference between AC-3_d and the new algorithms is that if the selected arc (v, w) and the reverse arc (w, v) are present in the queue then the new algorithms do not always compute the row support and the column support one after another as in AC-3_d. In AC-3_d double support heuristic is only used when both the arc (v, w) and the reverse arc (w, v) are in the queue. In AC-3_{dl} and in AC-3_{ds} irrespective of whether the reverse arc is present or not, they always use their own version of double support domain heuristic to compute the row support. The advantage is that if in the process of completely relaxing v the domain of v changes then arcs of the form (w, v) are added in the queue. This allows to compute the column support efficiently for arcs of the form (v, w) when reverse arcs of the form (w, v) are put in the queue *after*, relaxing $D(v)$ against $D(w)$, but were not in the queue when the $D(v)$ was relaxed against $D(w)$.

The only constant used by AC-3_{dl} is $unsupported$. AC-3_{ds} uses constants $single$, $double$, $unsupported$, and $support\_deleted$. All the constants are smaller than the values in the domains of the variables and are pairwise different. Both algorithms use two temporary arrays $rsupp[\,\cdot\,][\,\cdot\,]$ and $csupp[\,\cdot\,][\,\cdot\,]$ whose first dimension is bounded by n, and whose second dimension is bounded by d. For each arc (v, w), for each value $r \in D(v)$, $rsupp[\,w\,][\,r\,]$ records the value $c \in D(w)$ that provides support for r and similarly $csupp[\,w\,][\,c\,]$ records the value $r \in D(v)$ that provides support for c. AC-3_{ds} uses one more two dimensional array $rkind[\,\cdot\,][\,\cdot\,]$ which is used to remember what kind of support check resulted in a row support for the values in $D(v)$. The space complexity of all three data structures is $\mathcal{O}(n\,d)$. Both algorithms inherit $O(e + n\,d)$ space complexity and $\mathcal{O}(e\,d^3)$ time complexity from AC-3.

4.2 AC-3_{dl}

AC-3_{dl}'s algorithm for computing row support is shown in Figure 4. For a given arc (v, w), it tries to find a support for each value $r \in D(v)$ in $D(w)$ in a lexicographical order. For each $r \in D(v)$, $rsupp[\,w\,][\,r\,]$ records the first known value $c \in D(w)$ such that c supports r. Here a double-support check occurs if the support status of the first such value $c \in D(w)$ supporting r is unknown. In this case $csupp[\,w\,][\,c\,]$ is set to r. If the support status of c is already known then a single support-check occurs. If r fails to find a support in $D(w)$ then the value c of each previous neighbour w already known to support this deleted value r in $D(v)$ is marked *unsupported*.

```
function compute_row_support(v, w, var change_v):
begin
  change_v := False;
  for each c ∈ D(w) do begin
    csupp[ w ][ c ] := unsupported;
  end;
  for each r ∈ D(v) do begin
    if ∃c ∈ D(w) s.t. c supports r then
      rsupp[ w ][ r ] := first such value c;
      if csupp[ w ][ c ] = unsupported then
        csupp[ w ][ c ] := r;
      fi;
    else
      D(x) := D(x)\{ r };
      change_v := true;
      for each k such that row support of (v, k) is already computed do begin
        if csupp[ k ][ rsupp[ k ][ r ] ] = r then
          csupp[ k ][ rsupp[ k ][ r ] ] := unsupported;
        fi;
      end;
    fi;
  end;
end;
```

Fig. 4. Row support for AC-3_{dl}

```
function compute_column_support(w, v, var change_w):
begin
  change_w := False;
  for each c ∈ D(w) do begin
    if csupp[ w ][ c ] = unsupported then
      if ∄r ∈ D(v) s.t. rsupp[ w ][ r ] = c or rsupp[ w ][ r ] < c and r supports c then
        D(w) := D(w)\{c};
        change_w := True;
      fi;
    fi;
  end;
end;
```

Fig. 5. Column support for AC-3_{dl}

AC-3_{dl}'s algorithm for computing column support is shown in Figure 5. In AC-3_{dl} column support is computed only for those values of w that did not provide support for values in $D(v)$ or values whose support was deleted while computing the row support for other arcs of the form (v, w). As shown in the algorithm, for each $c \in D(w)$ whose support status is *unsupported*, it tries to find a support $r \in D(v)$ such that $rsupp[\,w\,][\,r\,] = c$ or $rsupp[\,w\,][\,r\,] < c$ and r supports c.

4.3 AC-3_{ds}

AC-3_{ds}'s algorithm for computing row support is shown in Figure 6. It tries to find a support for each value $r \in D(v)$ in $D(w)$ in a lexicographical order. When it tries to find a support for r it first uses double support checks and then single support checks until the support status of r is known. If it fails to find a support for r then the value c of each

```
function compute_row_support(v, w, var change_v):
begin
  change_v := False;
  for each c ∈ D(w) do begin
     csupp[ w ][ c ] := unsupported;
  end;
  for each r ∈ D(v) do begin
     if ∃c ∈ D(w) s.t. csupp[ w ][ c ] = unsupported and c supports r then
        rsupp[ w ][ r ] := first such value c;
        csupp[ w ][ rsupp[ w ][ r ] ] := r;
        rkind[ w ][ r ] := double;
     else if ∃c ∈ D(w) s.t. csupp[ w ][ c ] ≠ unsupported and c supports r then
        rsupp[ w ][ r ] := first such value c;
        rkind[ w ][ r ] := single;
     else
        D(x) := D(x)\{r};
        change_v := True;
        for each k such that row support of (v, k) is already computed do begin
           if rkind[ k ][ r ] = double then
              csupp[ k ][ rsupp[ k ][ r ] ] := support_deleted;
           fi;
        end;
     fi;
  end;
end;
```

Fig. 6. Row support for AC-3_{ds}

previous neighbour w (such that row support (v, w) is already computed) supporting this deleted value r in $D(v)$ is marked $support\_deleted$.

AC-3_{ds}'s algorithm for computing column support is depicted in Figure 7. In AC-3_{ds} column support is computed only for those values of w that did not provide support for values in $D(v)$ or values whose support was deleted while computing the row support for other arcs of the form (v, w). As shown in the algorithm column support is computed only for those values of w whose support status is *unsupported* or $support\_deleted$.

For each value $c \in D(w)$ whose support status is *unsupported*, the algorithm tries to seek support in $D(v)$ in exactly the same fashion as in AC-3_d while computing the column support. For each value $c \in D(w)$ whose support status is $support\_deleted$, it tries to seek a support in $r \in D(w)$ such that $rsupp[\,w\,][\,r\,] = c$ or r supports c.

5 Experimental Results

5.1 Introduction

In this section we will compare AC-3, AC-3_d, AC-3_{dl}, and AC-3_{ds}. We will measure their performance in terms of the CPU time in seconds, the number of support checks (checks), the number of times a revision ordering heuristic selects an element from the queue (selections), and the number of times a queue is updated (updates). As pointed out earlier, our goal is not to compare against optimal arc consistency algorithms. All algorithms were implemented in C. The experiments were carried out on a PC Pentium III having 256 MB of RAM running at 2.266 GHz processor with linux. Previously, statistical analysis of experimental data indicated that there is a significant and almost perfect linear relationship between the solution times (as well as checks) of any two arc

```
function compute_column_support(w, v, var change_w):
begin
  change_w := False;
  for each c ∈ D(w) do begin
     if csupp[ w ][ c ] = unsupported then
       if ∄r ∈ D(v) s.t. rsupp[ w ][ r ] < c and rkind[ w ][ r ] = double and r supports c then
         D(w) := D(w)\{c};
         change_w := True;
       fi;
     else if csupp[ w ][ c ] = support_deleted then
       if ∄r ∈ D(v) s.t. rsupp[ w ][ r ] = c or r supports c then
         D(w) := D(w)\{c};
         change_w := True;
       fi;
     fi;
  end;
end;
```

Fig. 7. Column support for AC-3_{ds}

consistency algorithms under consideration [14]. This justifies our decision to average the results for a particular class of problems over 50 runs.

For all the tables shown in this section, the column labelled as *heuristic* lists the revision ordering heuristics. [16] presents an implementation for reverse variable based heuristics facilitating $\mathcal{O}(n)$ selection for the optimal arc. All reverse variable based heuristics we consider in this section were implemented using that technique. Let *comp* be the variable selection order $\otimes_{\leq}^{\#} \bullet \otimes_{\geq}^{\delta_c} \bullet \otimes_{\leq}^{s}$, and let $comp_2$ be the variable selection order $\otimes_{\leq}^{\#} \bullet \otimes_{\geq}^{\delta_{cn}} \bullet \otimes_{\leq}^{s}$, where $\delta_{cn}(v)$ is the number of arcs of the form (v, w) currently present in the queue. The arc based heuristic *arc*:*comp* is given by $\otimes_{comp}^{\pi_2} \bullet \otimes_{comp}^{\pi_1}$, the variable based heuristic *var*:*comp* is given by $\otimes_{\leq}^{\#\circ\pi_1} \bullet \otimes_{comp}^{\pi_2}$, the reverse variable based heuristic *rev*:*comp* is given by $\otimes_{comp}^{\pi_2} \bullet \otimes_{comp}^{\pi_1}$, and the reverse variable based heuristic $rev : comp_2$ is given by $\otimes_{\leq}^{\#\circ\pi_2} \bullet \otimes_{\leq}^{s\circ\pi_2} \bullet \otimes_{comp_2}^{\pi_1}$. For algorithms such as MAC-3_d, MAC-3_{dl} and MAC-3_{ds} the order of processing reverse arcs is not important because they can not lead to a wipe out.

5.2 Stand-alone Arc Consistency

For stand alone arc consistency we experimented with random problems. They were generated by Frost *et al.*'s model B generator [7], which may be downloaded from `http://www.lirmm.fr/~bessiere/generator.html`. In this model a random CSP instance is characterised by (n, d, e, t) where n is the number of variables, d the uniform domain size, e the number of constraints, and t the number of no-goods. The problem classes we consider are P3 $= \langle 150, 50, 500, 2296\rangle$, and P4 $= \langle 50, 50, 1225, 2188\rangle$, which were also studied in [2, 3, 18]. Both classes correspond to problems in the phase transition [6]. P3 correspond to sparse and P4 correspond to dense problems. For random problems checks are implemented as cheap lookup operations. Tables 1 and 2 present the results for the before mentioned random problems.

Note that for a variable based heuristic, the number of selections of an element from the queue is very low. This is because when a variable is selected from the queue it allows to perform a number of revisions which is between 1 and the current degree of the variable. On the other side for an arc based heuristic, the number of selections of an element from the queue is high. The reason is that each arc that is selected from the queue corresponds to exactly one revision. However for AC-3_d it is somewhat reduced because sometimes AC-3_d performs two revisions per selection.

Table 1. Average results for random problems P3

algorithm	heuristic	checks	cpu-time	selections	updates
AC-3	arc:$comp$	2,449,084	0.033	5,956	1,353
AC-3	var:$comp$	3,885,849	0.047	1,123	1,891
AC-3	rev:$comp$	1,940,496	0.028	4,762	842
AC-3_d	arc:$comp$	1,888,750	0.032	4,628	1,415
AC-3_d	rev:$comp$	1,728,286	0.031	4,203	1,207
AC-3_{dl}	rev:$comp$	1,762,433	0.029	3,855	1,157
AC-3_{dl}	rev:$comp_2$	1,749,674	0.028	3,817	1,152
AC-3_{ds}	rev:$comp$	1,557,343	0.041	3,855	1,157
AC-3_{ds}	rev:$comp_2$	1,544,129	0.040	3,817	1,152

Table 2. Average results for random problems P4

algorithm	heuristic	checks	cpu-time	selections	updates
AC-3	arc:$comp$	5,454,546	0.076	16,318	573
AC-3	var:$comp$	6,139,919	0.080	375	1,009
AC-3	rev:$comp$	4,122,512	0.059	12,385	378
AC-3_d	arc:$comp$	3,609,732	0.070	10,777	575
AC-3_d	rev:$comp$	3,393,907	0.068	10,097	487
AC-3_{dl}	rev:$comp$	3,760,588	0.061	9,556	494
AC-3_{dl}	rev:$comp_2$	3,647,776	0.059	9,194	493
AC-3_{ds}	rev:$comp$	3,227,043	0.091	9,556	494
AC-3_{ds}	rev:$comp_2$	3,109,337	0.088	9,194	493

Remember that for a reverse variable based heuristic, if a variable v is selected from the queue it allows to perform revisions related to the number of arcs of the form (v, w) currently present in the queue. It is interesting to note that algorithms based on reverse variable based heuristics do not update their queue as frequently as variable based and arc based heuristics do. Notice that for both random problems, AC-3 with rev:$comp$ requires fewer checks and less time than it does for AC-3 with arc:$comp$.

We observe that AC-3 with rev:$comp$ and AC-3_{dl} with rev:$comp$ and rev:$comp_2$ appear to be faster in terms of the CPU time. We also observe that AC-3_{ds} despite of spending fewer support checks takes more time. This is probably because its domain heuristic is more expensive when compared to AC-3_d and AC-3_{dl}. It is not a coincidence that AC-3_{dl} and AC-3_{ds} when equipped with the same revision ordering heuristic always results in the same number of selections and updates. The reason is that the only difference between them is their domain heuristic.

5.3 Maintaining Arc Consistency during Search

In this section we will focus on the performance of the arc consistency algorithms during search (MAC [12]). Here we experimentally compare MAC-3, MAC-3_d, MAC-3_{dl}, and MAC-3_{ds} for solving random and real-world problems. During search all MACs visited the same nodes in the search tree. They were equipped with a *dom/deg* variable ordering heuristic with a lexicographical tie breaker where *dom* is the domain size and *deg* is the original degree of a variable.

Table 3. Average Results for random problems of size 25

algorithm	heuristic	checks	cpu-time	selections	updates
MAC-3	$arc{:}comp$	8,699,550	0.617	1,409,820	150,198
MAC-3	$var{:}comp$	8,316,780	0.304	79,539	241,948
MAC-3	$rev{:}comp$	7,704,453	0.499	1,192,148	109,258
MAC-3_d	$arc{:}comp$	5,747,741	0.526	848,161	151,270
MAC-3_d	$rev{:}comp$	5,549,216	0.453	803,162	108,209
MAC-3_{dl}	$rev{:}comp$	6,668,505	0.503	779,265	111,696
MAC-3_{dl}	$rev{:}comp_2$	6,614,958	0.493	768,942	109,983
MAC-3_{ds}	$rev{:}comp$	5,599,575	0.557	779,265	111,696
MAC-3_{ds}	$rev{:}comp_2$	5,546,977	0.546	768,942	109,983

Problems were generated for size $n \in \{15,20,25\}$, n values per domain $(n = d)$. The problems were generated as follows. Both tightness T and density C vary from 5% to 95% in 5% steps. For each combination of (C,T) 50 random problems were generated. The results for the average number of checks, the average solution time, the average number of selections and the average number of queue updates for a particular combination of algorithm and revision ordering heuristic for the problem size 25 is shown in Table 3.

MAC-3's checks for $arc{:}comp$ and $var{:}comp$ are about equal but it requires about 2.02 times less time for $var{:}comp$ than it does for $arc{:}comp$. MAC-3 with $rev{:}comp$ requires about 1.23 times less time than MAC-3 with $arc : comp$. It is interesting to note that MAC-3 with $var : comp$ requires about 17.02 times fewer selections and only 1.61 times more updates than MAC-3 with $arc : comp$. This may explain why MAC-3 with $var{:}comp$ is better when it comes to saving time. Again MAC-3_{ds} is the best among all lightweight arc consistency algorithms that we have considered when it comes to saving checks. The next best combination of algorithm and heuristic that saves time after MAC-3 with $var{:}comp$ is MAC-3_d with $rev{:}comp$.

Finally we present results for real world problems which came from the CELAR suite [4]. We did not consider optimisation but satisfiability only. Tables 4, 5 and 6 correspond to the results of RLFAP#5, RLFAP#11 and GRAPH#14 respectively. Checks were implemented as function calls for the real world problems.

The results in Tables 4, 5 and 6 show that again MAC-3_{ds} is the best when it comes to saving checks. Though MAC-3_{ds} spends fewer checks it does not always save time.

Table 4. Average results for real-world problem RLFAP 5

algorithm	heuristic	checks	cpu-time	selections	updates
MAC-3	$arc{:}comp$	5,567,209	0.935	1,292,826	530,589
MAC-3	$var{:}comp$	12,380,945	0.893	554,962	794,521
MAC-3	$rev{:}comp$	5,408,523	0.840	1,281,564	367,382
MAC-3_d	$arc{:}comp$	4,827,174	0.934	1,196,801	530,505
MAC-3_d	$rev{:}comp$	4,817,309	0.929	1,192,162	368,052
MAC-3_{dl}	$rev{:}comp$	4,817,750	0.849	1,148,423	367,981
MAC-3_{dl}	$rev{:}comp_2$	4,750,955	0.832	1,131,353	364,957
MAC-3_{ds}	$rev{:}comp$	4,489,648	0.869	1,148,423	367,981
MAC-3_{ds}	$rev{:}comp_2$	4,421,066	0.853	1,131,353	364,957

For RLFAP 5 and 11 MAC-3_{dl} with $rev{:}comp_2$ recorded the smallest solution time. For GRAPH 14 MAC-3_d with $rev{:}comp$ recorded the smallest solution time. MAC-3_{dl} and MAC-3_{ds} with $rev{:}comp_2$ perform better in terms of support checks, cpu-time, selections, and updates when compared to the same algorithms with $rev{:}comp$. Due to space restriction results for MAC-3 and MAC-3_d with $rev{:}comp_2$ are not shown but on average $rev{:}comp_2$ saves checks and (little) time when compared to $rev{:}comp$.

Despite using a lazy double support heuristic which does not always prefers double support checks, MAC-3_{dl} works well when compared to MAC-3_d whose double support domain heuristic $\mathcal{D}$ always prefers double support checks. As mentioned before it is not a coincidence that MAC-3_{dl} and MAC-3_{ds} when equipped with the same revision ordering heuristic always have the same number of selections and updates.

The results in Tables 4, 5, and 6 confirm that reverse variable based revision ordering heuristics save checks when compared to arc based and variable based revision ordering heuristics for a given algorithm. The results also confirm that they do not result in as many updates of the queue as with variable based and arc based heuristics. Also the number of selections is always less than arc based heuristics. This may be the reason why they always save time when compared to arc based heuristics.

Table 5. Average results for real-world problem RLFAP 11

algorithm	heuristic	checks	cpu-time	selections	updates
MAC-3	$arc{:}comp$	56,431,728	2.282	2,154,081	248,714
MAC-3	$var{:}comp$	52,510,653	1.641	151,358	539,820
MAC-3	$rev{:}comp$	43,957,986	1.590	1,654,675	163,826
MAC-3_d	$arc{:}comp$	30,810,434	1.685	1,168,121	248,999
MAC-3_d	$rev{:}comp$	30,801,235	1.642	1,163,567	161,758
MAC-3_{dl}	$rev{:}comp$	36,243,961	1.508	1,065,987	162,920
MAC-3_{dl}	$rev{:}comp_2$	35,575,214	1.494	1,084,724	150,947
MAC-3_{ds}	$rev{:}comp$	30,688,893	2.099	1,065,987	162,920
MAC-3_{ds}	$rev{:}comp_2$	29,995,844	2.093	1,084,724	150,947

Variable based heuristics require fewer selections than reverse variable and arc based heuristics. The overhead of selecting the best variable is limited for both variable and reverse variable based heuristics when compared to arc based heuristics.

Table 6. Average results for real-world problem GRAPH 14

algorithm	heuristic	checks	cpu-time	selections	updates
MAC-3	$arc{:}comp$	3,404,174	0.138	37,332	5,010
MAC-3	$var{:}comp$	3,399,796	0.112	4,493	4,922
MAC-3	$rev{:}comp$	3,051,426	0.106	32,942	3,470
MAC-3_d	$arc{:}comp$	1,744,372	0.106	25,704	4,984
MAC-3_d	$rev{:}comp$	1,723,346	0.089	25,223	3,487
MAC-3_{dl}	$rev{:}comp$	2,316,204	0.106	24,441	3,472
MAC-3_{dl}	$rev{:}comp_2$	2,309,559	0.102	24,269	3,470
MAC-3_{ds}	$rev{:}comp$	1,492,024	0.140	24,441	3,472
MAC-3_{ds}	$rev{:}comp_2$	1,484,398	0.139	24,269	3,470

6 Conclusion

We have classified revision ordering heuristics for arc consistency algorithms in three different categories: arc based, variable based, and reverse variable based heuristics. We pointed out advantages of using reverse variable based heuristics in terms of updating a queue and selecting an element from the queue. Experimental results demonstrate that algorithms using these heuristics are good in saving checks as well as time. Reverse variable based version of AC-3 was also discussed.

We presented two new lightweight arc consistency algorithms AC-3_{dl} and AC-3_{ds}. Both algorithms use reverse variable based heuristics. They only differ by their domain heuristic. Overall MAC-3_{dl} was good in saving time despite of using a lazy double support domain heuristic which does not always prefer double support checks. For all the real world problems and the random problems that we have considered AC-3_{ds} is the best when it comes to saving checks. But it is not always that good in saving time. This is probably because of its domain heuristic, which is more expensive when compared to the domain heuristics of AC-3_d and AC-3_{dl}.

There is no single winner when it comes to save time. For stand alone arc consistency, AC-3 with $rev{:}comp$ is good in saving checks and time when compared to AC-3 with $var{:}comp$ and $arc{:}comp$. For search, MAC-3 with $var{:}comp$ becomes the fastest solver for random problems. For real world problems that we considered, MAC-3_{dl} with $rev : comp_2$ is the quickest solver. In [16] we have shown that reverse variable based heuristics also save checks and time when used with coarse-grained heavyweight arc consistency algorithms such as AC-2001. MAC-3 with $var{:}comp$ is the worst in saving checks while MAC-3_{ds} with $rev{:}comp_2$ is the best. On average MAC-3_{ds} saves 50% support checks when compared to MAC-3 with $var{:}comp$.

References

1. C. Bessière and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the* 11^{th} *National Conference on Artificial Intelligence (AAAI'93)*, Washington, DC, 1993.
2. C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers.
3. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 309–315, 2001.
4. B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Journal of Constraints*, 4:79–89, 1999.
5. 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.
6. 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.
7. I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.
8. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
9. 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.
10. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
11. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
12. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.
13. 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.
14. M.R.C. van Dongen. Improving MAC by sacrificing the optimality of the worst case time-complexity of its arc-consistency component, 2004. In preparation.
15. M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 2004. Accepted for publication.
16. M.R.C. van Dongen and D. Mehta. Queue representation for arc consistency algorithms, 2004. Submitted for publication.
17. 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.
18. Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 316–321, 2001.