Reducing Checks and Revisions in Coarse-grained MAC Algorithms[*]

D. Mehta[†] and M.R.C. van Dongen
Cork Constraint Computation Centre, Ireland

Abstract

Arc consistency algorithms are widely used to prune the search space of Constraint Satisfaction Problems (CSPs). Coarse-grained arc consistency algorithms like AC-3, AC-3$_d$ and AC-2001 are efficient when it comes to transforming a CSP to its arc-consistent equivalent. These algorithms repeatedly carry out *revisions*. Revisions require *support checks* for identifying and deleting all unsupported values from the domain of a variable. In revisions for difficult problems most values have *some* support. Indeed, most revisions are *ineffective*, i.e. they cannot delete any value and consume a lot of checks and time. We propose two solutions to overcome these problems. First we introduce the notion of a *Support Condition* (SC) which guarantees that a value has *some* support. SCs reduce support checks while maintaining arc consistency during search. Second we introduce the notion of a *Revision Condition* (RC) which guarantees that *all* values have support. A RC avoids a candidate revision and queue maintenance overhead. For random problems, SCs reduce the checks required by MAC-3 (MAC-2001) up to 90% (72%). RCs avoid at least 50% of the total revisions. Combining the two results in reducing 50% of the solution time.

1 Introduction

Arc consistency (AC) algorithms are widely used to prune the search space of binary Constraint Satisfaction Problems (CSPs). MAC [Sabin and Freuder, 1994] is a backtrack algorithm that Maintains Arc Consistency during search. It reduces the thrashing behaviour of a backtrack algorithm, which usually fails many times as a result of the same local inconsistencies. MAC-*x* uses AC-*x* for maintaining arc consistency during search.

Many arc consistency algorithms have been proposed. On the one hand there are algorithms such as AC-3 [Mackworth, 1977] and AC-3$_d$ [van Dongen, 2004]. These algorithms have a low $\mathcal{O}(e + n\,d)$ space complexity but they repeat their support checks and have a non optimal bound of $\mathcal{O}(e\,d^3)$ for their worst-case time complexity. Here e is the number of constraints, n the number of variables and d the maximum domain size of the variables. On the other hand there are algorithms such as AC-4 [Mohr and Henderson, 1986], AC-6 [Bessière and Cordier, 1993], and AC-2001 [Bessière and Régin, 2001] that use auxiliary data structures to avoid repeating their support checks and have an optimal worst-case time complexity of $\mathcal{O}(e\,d^2)$.

Since the introduction of AC-4 [Mohr and Henderson, 1986] much work has been done to avoid repeating support checks by using auxiliary data structures. Depending upon the algorithm, these auxiliary data structures store one or more supports for each value involved in each constraint. The belief is that reducing checks helps in solving problems more quickly. However, allowing algorithms to repeat (not too many) checks, relieves them from the burden of a large additional bookkeeping and this may save time, especially if checks are cheap. In this paper, we introduce the notion of a *Support Condition* (SC) which guarantees that a value has *some* support. SCs help avoiding many (but not all) sequences of support checks eventually leading to a support *without storing and maintaining support values*.

Coarse-grained arc consistency algorithms repeatedly carry out revisions to remove all unsupported values from the domain of a variable. Many revisions are *ineffective*, i.e. they cannot remove any value. For example, when RLFAP #11 is solved using MAC-3 or MAC-2001, 83% of the total revisions are ineffective. We introduce the notion of a *Revision Condition* (RC) which guarantees that *all* values have some support. RCs help avoiding many (but not all) ineffective revisions and much queue maintenance, which is also a great source of time consumption [van Dongen and Mehta, 2004]. Furthermore, we show that reverse variable based heuristics [Mehta and van Dongen, 2004] result in fewer revisions.

The remainder of this paper is as follows. Section 2 is a brief introduction to constraints. Section 3 discusses revision ordering heuristics and the coarse-grained algorithm AC-3. Sections 4 and 5 introduce the notions of a support condition and a revision condition. Section 6 presents experimental results. Conclusions are presented in Section 7.

[*]This work has received some support from Science Foundation Ireland under Grant No. 00/PI.1/C075.

[†]The first author is supported by Boole Centre for Research in Informatics.

2 Constraint Satisfaction

A *Constraint Satisfaction Problem* is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x)$ for each variable $x \in \mathcal{V}$ and a set of e constraints among subsets of variables of $\mathcal{V}$. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of $D(x)$ and $D(y)$ that specifies the allowed pairs of values for x and y. We only consider CSPs whose constraints are binary.

A value $b \in D(y)$ is called a *support* for $a \in D(x)$ if $(a, b) \in C_{xy}$. Similarly $a \in D(x)$ is called a support for $b \in D(y)$ if $(a, b) \in C_{xy}$. A *support check* (consistency check) is a test to find if two values support each other. A value $a \in D(x)$ is called *viable* if for every variable y constraining x the value a is supported by some value in $D(y)$. A CSP is called *arc-consistent* if for every variable $x \in \mathcal{V}$, each value $a \in D(x)$ is viable.

The *density* p_1 of a CSP is defined as $2\,e/(n^2 - n)$. The *tightness* p_2 of the constraint C_{xy} between the variables x and y is defined as $1 - |\,C_{xy}\,|/|\,D(x) \times D(y)\,|$. The *degree* of a variable is the number of constraints involving that variable. Before starting search MAC transforms the input CSP to its arc-consistent equivalent. We call the domain of a variable in this arc-consistent equivalent as the *arc-consistent* domain of that variable. For the remainder of this paper for any variable x, we use $D_{ac}(x)$ for the arc-consistent domain of x, and $D(x)$ for the current domain of x. The *directed constraint graph* of a given CSP is a directed graph having an arc (x, y) for each combination of two mutually constraining variables x and y. We will use G to denote the directed constraint graph of the input CSP.

3 Revision Ordering Heuristics

Coarse-grained arc consistency algorithms use *revision ordering heuristics* to select an arc from a data structure called a queue (set really). When an arc, (x, y), is selected from the queue, $D(x)$ is *revised* against $D(y)$. Here to *revise* $D(x)$ against $D(y)$ means removing all values from $D(x)$ that are not supported by any value of $D(y)$. Revision ordering heuristics can influence the efficiency of arc consistency algorithms [Wallace and Freuder, 1992]. They can be classified into three categories: *arc based*, *variable based* [McGregor, 1979], and *reverse variable based* [Mehta and van Dongen, 2004] heuristics. The differences between them are as follows.

Arc based revision ordering heuristics are the most commonly presented. Given some criterion they select an arc (x, y) for the next revision. Selecting the best arc can be expensive [van Dongen and Mehta, 2004]. Each selected arc corresponds to exactly one revision. Since there may be many revisions there may be many selections and this may result in a significant overhead. When arc based heuristics are used, the queue needs to be updated after every effective revision unless the domain of a variable becomes empty. Many updates can be an overhead too.

Variable based heuristics [McGregor, 1979] first select a variable x and then repeatedly select arcs of the form (y, x) for the next revision until no more such arcs exist or some $D(y)$ becomes empty. They may be regarded as *propagation based heuristics* because the consequences of removing one or more values from $D(x)$ are propagated in all of its neighbours. If a variable x is selected then the domain of all its neighbours in the constraint graph will be revised against $D(x)$. In this setting there are usually fewer selections from the queue at the cost of performing more checks. If checks are cheap then time can be saved because the queue needs fewer selections. Here too every effective revision results in updating the queue.

Reverse variable based heuristics [Mehta and van Dongen, 2004] first select a variable x and then repeatedly select arcs of the form (x, y) for the next revision until there are no more such arcs or $D(x)$ becomes empty. They may be regarded as *support based heuristics* because for one variable x at a time, each value in $D(x)$ seeks support with respect to all of its neighbours for which it is currently unknown whether such support exists. When a variable x is selected a number of revisions is performed which is between 1 and the number of arcs of the form (x, y) currently present in the queue. Therefore, the number of selections (of x), and the overhead of queue management, is usually less.

Selecting a variable x and revising it against all its neighbours y such that (x, y) is currently present in the queue we call a *complete relaxation* of x. Another advantage of using reverse variable based heuristics is that the queue only needs to be updated after every effective *complete relaxation* and not after every effective *revision*, which reduces the number of times the arcs are added to the queue. Overall fewer revisions are performed which results in saving support checks. Pseudo-code for AC-3 equipped with reverse variable based revision ordering heuristic is depicted in Figure 1. The revise function upon which AC-3 depends is depicted in Figure 4.

```
**Function** AC-3: **Boolean**;
**begin**
  $Q := G$
  **while** $Q$ not empty **do begin**
    select any $x$ from $\{ x \; : \; (x, y) \in Q \}$
    $effective\_revisions := 0$
    **for each** $y$ such that $(x, y) \in Q$ **do**
      remove $(x, y)$ from $Q$
      $revise(x, y, change_x)$
      **if** $D(x) = \emptyset$ **then**
        **return** False
      **else if** $change_x$ **then**
        $effective\_revisions := effective\_revisions + 1$
        $y'' := y$;
    **if** $effective\_revisions = 1$ **then**
      $Q := Q \cup \{\, (y', x) \mid y'$ is a neighbour of $x \wedge y' \neq y'' \}$
    **else if** $effective\_revisions > 1$ **then**
      $Q := Q \cup \{\, (y', x) \mid y'$ is a neighbour of $x \}$
  **return** True;
**end**;
```

Figure 1: AC-3.

In Figure 1, if $D(x)$ was changed after a complete relaxation and if this was the result of *only one* effective revision (*effective_revisions* $= 1$), which happened to be against $D(y'')$, then all arcs of the form (y', x) where y' is a neighbour of x and $y' \neq y''$ are added to the queue. However, if $D(x)$ was changed as the result of *more than one* effective revision (*effective_revisions* > 1) then *all* arcs of the form (y', x) where y' is a neighbour of x are added to the queue. Modulo constraint propagation effects this avoids queue maintenance overhead.

4 A Support Condition

Arc consistency algorithms are based on the notion of support. Most arc consistency algorithms proposed so far put a lot of effort in *identifying* a support to confirm the existence of a support. Identifying the support is more than is needed to guarantee that a value is supportable: knowing that a support exists is enough. Optimal algorithms like AC-2001 and AC-6 always keep track of the last known support for each value. When this support is lost they try to identify the next support. AC-4 is the only algorithm that confirms the existence of a support by not identifying it during the course of search but its inefficiency lies in its space complexity $\mathcal{O}(e\,d^2)$ and the necessity of maintaining huge data structures during search.

We propose the notion of a *support condition* (SC), which guarantees that a value has some support. The key point is that it guarantees the existence of a support without identifying it and without storing and maintaining support values. Let C_{xy} be the constraint between x and y, let $a \in D(x)$ (also denoted as (x, a)) and let *scount*$[\,x, y, a\,]$ be the number of supports of (x, a) in $D_{ac}(y)$. A value $a \in D(x)$ is supported by y if it has at least one support in $D(y)$. Furthermore if $R(y) = D_{ac}(y) \setminus D(y)$ are the removed values from the arc-consistent domain of y then $|\,R(y)\,|$ is an upper bound on the number of lost supports of (x, a) in y. Therefore, if the following condition is true then (x, a) is supported by y:

$$\textit{scount}[\,x, y, a\,] > |\,R(y)\,| \qquad (1)$$

In order to use the Equation (1) in any coarse-grained MAC algorithm, for each arc-value pair involving the arc (x, y) and the value a on x, $scount[\,x, y, a\,]$ must be assigned the number of supports of (x, a) in $D_{ac}(y)$. Once these support counters are initialised they remain static during search. Hence, there is no overhead of maintaining them. The pseudo-code for computing the support count for each arc-value pair is depicted in Figure 2. In the algorithm, *last*$[\,x, y, a\,]$ stores AC-2001's last known support for (x, a) in y. Note that the algorithm does not repeat checks and uses the bidirectional property of constraints. For easy problems initialising support counters can be an overhead in terms of support checks. However, it can save time and checks for hard problems.

Next we generalise the idea presented in the Equation (1). Here we associate a weight (non-negative integer) $w[\,x, y, a\,]$ with each arc-value pair. If the following condition is true

```
Function InitialiseSupportCounters ( )
  call AC-2001
  if the problem is arc-consistent then
    for each $(x, y) \in G$ do
        for each $a \in D_{ac}(x)$ do
            $scount[\,x, y, a\,] := 1$
    for each $(x, y) \in G$ such that $x < y$ do
        for each $a \in D_{ac}(x)$ do
            for each $b \in D_{ac}(y)$ such that $b > last[\,x, y, a\,]$ do
                if $b$ supports $a$ then begin
                    $scount[\,x, y, a\,] := scount[\,x, y, a\,] + 1$
                    $scount[\,y, x, b\,] := scount[\,y, x, b\,] + 1$
                end
```

Figure 2: Initialisation of support counters.

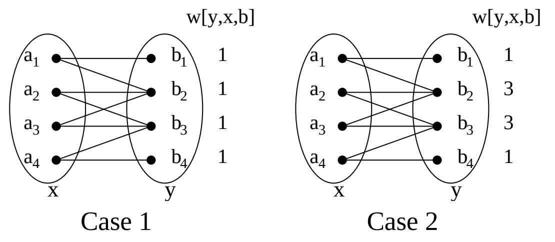

Figure 3: Support inference using different weights

then (x, a) is supported by y:

$$\sum_{(a,b)\,\in\, C_{xy}} w[\,y, x, b\,] \;>\; \sum_{b'\,\in\, R(y)} w[\,y, x, b'\,]\,. \qquad (2)$$

We call the left hand side of Equation (2) the *cumulative weight* of (x, a) with respect to y: it is equal to the sum of the weights of the supports of (x, a) in $D_{ac}(y)$. We call the right hand side of Equation (2) the *removed weight* from $D_{ac}(y)$ with respect to x: it is equal to the sum of the weights of the values removed from $D_{ac}(y)$ with respect to x. Note that if the weight associated with each arc-value pair is 1 then Equation (1) is a special case of the condition in Equation (2). In that case the cumulative weight of (x, a) with respect to y is equal to the number of supports of (x, a) in $D_{ac}(y)$ and the removed weight from $D_{ac}(y)$ is equal to the number of values removed from $D_{ac}(y)$. We call the condition in Equation (2) a *Support Condition* (SC). A SC guarantees the existence of a support and may allow to save many checks.

We illustrate the use of SC to infer the existence of a support by using the example as shown in Figure 3. In Figure 3 (Case 1), the weight associated with each value is 1. The cumulative weight of a_1 with respect to y is 2: it is equal to the number of the supports of a_1 in $D_{ac}(y)$. Now assume that b_1 is deleted from $D_{ac}(y)$ then the removed weight from $D_{ac}(y)$ with respect to x is 1: it equals $|\,R(y)\,|$. It can be inferred that a_1 still has a support in $D(y)$ since the cumulative weight of a_1 is greater than the removed weight from $D_{ac}(y)$. Similarly, a support for any value in $D(x)$ can be inferred if any single value is removed from $D_{ac}(y)$. If any two values are removed from $D_{ac}(y)$ then the removed weight from $D_{ac}(y)$ is 2. Here a support cannot be inferred for any value of $D(x)$ because the cumulative weight of each value in $D(x)$ is 2, which is not greater than the removed weight.

If other weights are used and if SC holds for any value $a \in D(x)$ then a support is still guaranteed to exist in $D(y)$. Let us examine Case 2 of Figure 3 where the weight assigned to each value in $D_{ac}(y)$ is its own support count with respect to x. Now the cumulative weight of a_1 is 4 since the weights of the supports of b_1 and b_2 are 1 and 3 respectively. Like Case 1, if any single value is removed from $D_{ac}(y)$, a support can be inferred for all values in $D(x)$. If the two values b_1 and b_4 are removed then the removed weight from $D_{ac}(y)$ is 2. Unlike Case 1 a support can be inferred for all the values of x. In another situation if b_1 and b_2 are removed then also the existence of support can be inferred for at least a_2 and a_3 since their cumulative weights are 6 and the removed weight from $D_{ac}(y)$ is 4. We will see further on that using support

```
Function $revise(x, y, \textbf{var}\ change_x)$
begin
    $change_x :=$ False
    for each $a \in D(x)$ do
        if $\sum_{(a,b) \in C_{xy}} w[y, x, b] > \sum_{b' \in R(y)} w[y, x, b']$ then
            skip /* $a$ is supported */
        else if $\nexists b \in D(y)$ such that $b$ supports $a$ then begin
            $D(x) := D(x) \setminus \{ a \}$
            $change_x :=$ True
        end
end;
```

Figure 4: Algorithm *revise* of AC-3.

counts as weights lets SCs save more checks.

The *revise* function for AC-3 is depicted in Figure 4. This function is slightly different from its original version because it uses SC to avoid a series of checks for which it can be known in advance that it will eventually lead to a support. The use of SC is not restricted to AC-3. It can be integrated in any coarse-grained algorithm.

5 A Revision Condition

The support check is the core operation carried out by arc consistency algorithms. To reduce the number of checks, algorithms proposed so far (1) perform viability checks like in AC-2001, (2) check the status of counters like in AC-4, (3) make some inference based on the supports stored in auxiliary data structures like in AC-7, or (4) carry out SCs as mentioned in Section 4. We call all these tests *auxiliary support checks* (ASCs). When support checks are not too expensive then ASCs can be an overhead and reducing the checks *alone* does not always help in reducing the solution time.

All the improvements proposed so far to reduce support checks are done at fine level of granularity. We propose a *coarser check* at arc level. The idea is that we will use this coarser check to avoid a complete revision. This will not only save support checks but will also avoid *auxiliary* support checks. For a given arc (x, y), if the least cumulative weight (LCW) of the values of $D(x)$ with respect to y is greater than the removed weight from $D_{ac}(y)$ then all the values of $D(x)$ are supported by y. This can be expressed as follows:

$$\min \left\{ \sum_{(a,b) \in C_{xy}} w[y, x, b] : a \in D(x) \right\} > \sum_{b' \in R(y)} w[y, x, b'] \,. \tag{3}$$

We call this condition a Revision Condition (RC). The RC can be used by all coarse-grained arc consistency algorithms to reduce unnecessary revisions while maintaining full arc consistency during search.

If a RC holds then it can be exploited *after* selecting the arc (x, y) or when arcs are added to the queue. In the former case the corresponding revision is not carried out and in the latter case the arc (x, y) is not added to the queue. We will use the revision condition by tightening the condition for adding the arcs to the queue: arcs should only be added if RC does not hold. This is depicted in Figure 5. With this implementation the advantages of using the RC are threefold: (a) it will reduce the number of arcs that have to be added to the queue, (b) it will reduce the number of arcs that have to be selected from the queue, and (c) it will reduce the total number of revisions.

```
if $effective\_revisions = 1$ then
  $Q := Q \cup \{ (y', x) \mid y'$ is a neighbour of $x \wedge y' \neq y'' \wedge$
  $\min\{ \sum_{(a,b) \in C_{y'x}} w[x, y', b] : a \in D(y')\} \leq \sum_{b' \in R(y')} w[x, y', b']\}$
else if $effective\_revisions > 1$ then
  $Q := Q \cup \{ (y', x) \mid y'$ is a neighbour of $x \wedge$
  $\min\{ \sum_{(a,b) \in C_{y'x}} w[x, y', b] : a \in D(y')\} \leq \sum_{b' \in R(y')} w[x, y', b']\}$
```

Figure 5: Enforcing RC while adding the arcs to the queue.

Note that both SC and RC are presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. We compute the cumulative weight for each arc-value pair before the search starts. Whenever a value is deleted from the domain of a variable x, we update the removed weight for all the arcs of the form $(\cdot, x)$. If there is a change in the domain size of a variable x after a complete relaxation, we update the LCW for all the arcs of the form $(x, \cdot)$. The space complexity of storing the cumulative weights is $\mathcal{O}(e\,d)$. The space complexity of storing the removed weights and the LCWs for all the arcs is $\mathcal{O}(e)$ but it may increases to $\mathcal{O}(e\,n)$ during search. The overall space complexity of using SC in conjunction with RC is $\mathcal{O}(e\,d + e\,n) = \mathcal{O}(e\,max(d, n))$.

Updating the least cumulative weight for all the arcs of the form $(x, \cdot)$, if there is a change in the domain of x after every complete relaxation can be an overhead. The other option is to update the LCW of only one arc (x, y) while revising $D(x)$ against $D(y)$ instead of all the arcs of the form $(x, \cdot)$ after a complete relaxation. This can be done cheaply and may avoid the addition of the arc (x, y) to the queue in future. This can be considered as a weak version of a revision condition (WRC) because the LCW is not maintained for all the arcs. The disadvantage here is that not all possible ineffective revisions will be saved then can be saved by RC.

Independent work by [Boussemart *et al.*, 2004] uses $\min\{ \sum_{(a,b) \in C_{xy}} 1 : a \in D_{ac}(x) \} > \sum_{b' \in R(y)} 1$, which is a special case of Equation (3). Note that in their setting $w[x, y, a]$ is 1 for each arc-value pair and the above condition is exploited *after* determining the arc for the revision. For a given arc (x, y), this condition examines the least cumulative weight (the least support count) of the values of $D_{ac}(x)$ and not of $D(x)$ with respect to y as is done in Equation (3). We call this condition a *static* version of a revision condition because the least cumulative weight is not updated for any arc during the course of search. It remains static for all the arcs. This may not allow to avoid as many ineffective revisions as can be avoided by RC.

6 Experimental Results

6.1 Introduction

In this section, we shall present some results to prove the practical efficiency of SC, RC and WRC. We implemented them in MAC-3 and MAC-2001 equipped with *comp* [van Dongen, 2004] as a revision ordering heuristic. We denote the arc based, reverse variable based, and the variable based heuristic *comp* as *arc:comp*, *rev:comp*, and *var:comp* respectively. The details about these heuristics can be found in

Heuristic	Condition	Weight	MAC-3		Revisions	MAC-2001	
			Checks	Time	Revisions	Checks	Time
	-	-	79,644,545	25.39	16,627,343	20,876,139	27.20
arc:comp	SC	1	10,226,932	23.22	16,627,343	7,122,238	25.52
	WRC	1	28,632,454	15.51	6,944,423	17,188,921	18.56
	SC + WRC	1	10,226,932	14.49	6,944,423	7,122,238	16.46
	-	-	74,985,422	21.76	15,297,543	20,587,469	22.88
	SC	1	10,056,858	19.34	15,297,543	7,064,437	21.23
	SC	*scount*	7,149,704	26.59	15,297,543	5,569,970	29.14
	WRC	1	28,136,221	13.44	6,708,679	17,063,043	15.80
	WRC	*scount*	18,548,689	16.21	4,641,513	12,614,384	19.04
rev:comp	SC + WRC	1	10,056,858	12.21	6,708,679	7,064,437	13.80
	SC + WRC	*scount*	7,149,704	15.48	4,641,513	5,569,970	17.44
	RC	1	22,256,549	12.74	5,153,349	13,971,627	21.73
	RC	*scount*	13,902,999	18.03	3,276,247	9,724,569	20.20
	SC + RC	1	10,056,858	12.82	5,153,349	7,064,437	21.45
	SC + RC	*scount*	7,149,704	18.22	3,276,247	5,569,970	20.27

Table 1: Results for the random problems.

Heuristic	Conditions	MAC-3		Revisions	MAC-2001	
		Checks	Time	Revisions	Checks	Time
	-	56,431,728	2.656	2,154,081	10,412,277	2.661
arc:comp	SC	36,832,553	2.370	2,154,081	15,217,649	2.626
	WRC	37,250,230	1.771	809,623	14,886,826	1.763
	SC + WRC	36,832,553	1.755	809,623	15,217,649	1.824
	-	42,519,506	1.931	1,602,603	10,332,998	2.082
	SC	28,429,473	1.750	1,602,603	15,147,054	2.114
rev:comp	WRC	28,829,358	1.380	752,277	14,835,844	1.243
	SC + WRC	28,429,473	1.382	752,277	15,147,054	1.270
	RC	28,818,246	1.448	750,789	14,825,485	1.450
	SC + RC	28,429,473	1.467	750,789	15,147,054	1.475

Table 2: Results for RLFAP#11.

[Mehta and van Dongen, 2004]. 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. All algorithms were written in C. The experiments were carried out on linux on a PC Pentium III (2.266 GHz processor and 256 MB RAM). Performance is measured in terms of the number of support checks, the CPU time in seconds, and the number of revisions.

We experimented with random problems which were generated by Frost *et al.*'s model B generator [Gent *et al.*, 2001] (`http://www.lirmm.fr/~bessiere/generator.html`). In this model a random CSP instance is characterised by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the uniform domain size, p_1 the average density, and p_2 the uniform tightness. For each combination of $\langle n, d, p_1, p_2 \rangle$, 50 random problems were generated. Table 1 shows the mean results for $\langle 50, 10, 1.0, 0.13 \rangle$ which is located at the phase transition. In Table 1, under the column labelled as Weight, 1 denotes that the weight associated with each arc-value pair is 1 (also the default value) while *scount* denotes that the weight associated with each arc-value pair is its own support count. Table 2 corresponds to the real world instance RLFAP #11 which came from the CELAR suite. Table 3 corresponds to a quasigroup with holes problem [Achlioptas *et al.*, 2000] of order 10 and 74 holes.

6.2 Discussion

One can immediately notice that SC reduces the number of checks required by MAC-3 and MAC-2001. For instance for the random problem, checks required by MAC-3 and MAC-2001 are reduced by 90% and 72% respectively, when the weight associated with each arc-value pair is its own support count. We observed that for the random problems SC and RC are especially effective for difficult problems that take a lot of time. The original version of MAC-3 requires 3.81 times more checks than the original version of MAC-2001 but it reduces to 1.28 after using SC. It is interesting to note that for the random problem and the quasigroup problem MAC-3 with SC requires fewer checks than the original version of MAC-2001. Remember that MAC-3 uses a non-optimal algorithm AC-3 while MAC-2001 uses an optimal algorithm AC-2001 but that they usually repeat checks in different branches of the search. In case of RLFAP #11 when SC is used in MAC-2001 there is an improvement in the number of checks required during the course of search but not in the total number of checks. This is caused by the initialisation of the cummulative weights.

MAC-2001 always spends fewer checks than MAC-3 but requires more time. This corresponds to the results presented in [van Dongen, 2004]. Using SC in MAC-3 and MAC-2001 reduces the number of checks but this is not reflected in the solution time. In fact there is only a marginal saving and in some cases it may consume more time. This shows that when checks are cheap carrying out ASCs to reduce the checks is not really a great help in reducing the overall solution time.

Both RC and WRC avoid at least 50% of the total revisions. RCs are able to save more ineffective revisions than WRCs. Satisfying a RC or a WRC avoids a complete revision that not only reduces checks but also ACSs and the overhead of queue management. When *rev:comp* and WRC are used together in MAC-3 or in MAC-2001 there is on average 50% reduction in the solution time compared to the original algorithms equipped with *arc:comp*, which is significant. Results for *arc:comp* with RC are not presented due to the space restriction. The overhead to maintain LCWs is less with *rev:comp* because they need to be updated after every effective complete relaxation but in case of *arc:comp* and *var:comp* after

		MAC-3			MAC-2001	
Heuristic	Conditions	Checks	Time	Revisions	Checks	Time
arc:comp	-	1,760,191,958	202.28	600,538,223	585,239,320	220.88
	SC	444,247,815	204.23	600,538,223	355,925,876	213.84
	WRC	564,731,731	117.20	237,506,329	453,353,800	129.54
	SC + WRC	444,247,815	120.37	237,506,329	355,925,876	129.70
rev:comp	-	1,705,703,347	184.61	573,319,558	571,663,383	203.77
	SC	445,253,641	185.66	573,319,558	356,614,116	198.47
	WRC	565,520,375	110.65	238,239,232	453,868,110	124.16
	SC + WRC	445,253,641	111.33	238,239,232	356,614,116	126.05
	RC	552,491,872	120.65	229,190,153	443,596,880	193.45
	SC + RC	445,253,641	125.97	229,190,153	356,613,391	193.90

Table 3: Results for Quasigroup problems of order 10 with 74 holes.

every effective revision. Results shown in all the tables confirm that *rev:comp* is good in saving revisions, support checks and the CPU time for both MAC-3 and MAC-2001 when compared to *arc:comp*.

Finally, we compared RC with the static version of RC. For the random problems when solved with MAC-3, the static version of RC with *var:comp* as implemented in [Boussemart *et al.*, 2004] spends on average $10,147,799$ revisions, $25,328,772$ checks and 10.86 seconds while RC with *rev:comp* spends on average spends $515,349$ revisions, $10,056,858$ checks and 12.74 seconds. The static version of RC spends about twice as many revisions and checks as RC but saves time. The reason for this is that checks are very cheap for random problems and that the static version uses a variable based queue, which has low maintenance overhead.

7 Conclusions

In this paper first we present a *support condition*. Satisfying SC guarantees the existence of a support for a value. SCs avoid many positive and negative support checks during the course of the search. We showed that when checks are cheap reducing them by using ASCs do not payoff a lot in terms of the CPU time. Instead of carrying out ASCs for each value we tried to reduce the number of ineffective revisions through *revision condition* and a *weak* version of revision condition. Both of them reduce the number of arcs that have to be added to the queue. Having fewer arcs in the queue improves the selection of the best arc from the queue. Furthermore fewer revisions are performed. Overall, SCs reduce the positive and negative support checks required by MAC-3 and MAC-2001. RCs avoid at least 50% of the total revisions. Combining the two results in reducing the solution time by 50%.

Acknowledgements

Thanks to Barbara Smith, Mark Hennessy, Richard Wallace and the other members of the Cork Constraint Computation Centre for helpful comments.

References

[Achlioptas *et al.*, 2000] Dimitri Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI/IAAI*, pages 256–261, 2000.

[Bessière and Cordier, 1993] 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.

[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 (IJCAI'2001)*, pages 309–315, 2001.

[Boussemart *et al.*, 2004] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.

[Gent *et al.*, 2001] 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.

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

[McGregor, 1979] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.

[Mehta and van Dongen, 2004] D. Mehta and M.R.C. van Dongen. Two new lightweight arc consistency algorithms. In M.R.C. van Dongen, editor, *Proceedings of the First International Workshop on Constraint Propagation and Implementation (CPAI'2004)*, pages 109–123, 2004.

[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 (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.

[van Dongen and Mehta, 2004] M.R.C. van Dongen and D. Mehta. Queue representation for arc consistency algorithms. In L. McGinty and B. B.Crean, editors, *Proceedings of the Fifteenth Irish Conference on Artificial Intelligence and Cognitive Science*, pages 334–343, 2004.

[van Dongen, 2004] M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3–4):317–334, 2004.

[Wallace and Freuder, 1992] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, B.C., 1992.