Constraint Programming Letters 2 (2007) 37–53 Submitted 4/2007; Published 1/2008

Reducing Checks and Revisions in the Coarse-grained Arc Consistency Algorithms

Deepak Mehta DM6@CSMAIL.UCC.IE
Computer Science Department
University College Cork
Cork, Ireland

Editor: M.R.C. van Dongen

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 and AC-2001 are efficient in establishing arc consistency on a given CSP. These algorithms repeatedly carry out *revisions*. Revisions require *support checks* for identifying and deleting all unsupported values from the domains. For difficult problems, many values find *some* support while revising domains. Indeed, many revisions are *ineffective*, that is 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). If the SC holds then it guarantees that a given value has *some* support. SCs reduce support checks while maintaining arc consistency during search. Second, we introduce the notion of a *Revision Condition* (RC). If the RC holds then it guarantees that *all* values of a given domain have some support. An RC avoids a candidate revision and queue maintenance overhead. Our experimental results show that for random problems, SCs reduce the support checks required by MAC-3 (MAC-2001) up to 90% (72%). The RCs avoid at least 50% of the total revisions. Combining the two, results in reducing 50% of the solution time.

Keywords: Constraint Satisfaction Problems, Filtering Algorithms, Arc Consistency.

1. Introduction

Maintaining Arc Consistency (MAC) is considered to be one of the most efficient generic approach in solving large and hard instances of Constraint Satisfaction Problems (CSPs) (Sabin and Freuder, 1994; Bessière and Régin, 1996). Each improvement in its underlying arc consistency algorithm can enhance the performance of the MAC algorithm significantly. Therefore, proposing efficient algorithms and techniques for enforcing arc consistency has been an interest of research from the last two decades.

Arc consistency algorithms are based on the notion of a support. Most of the algorithms proposed so far put a lot of effort in *identifying* a support to confirm the *existence* of a support. When a support is sought for a given value in a given domain, a sequence of support checks is usually performed. If a support exists it is identified, otherwise the value is deleted. Identifying the support is more than is needed to guarantee that a value is supportable: *knowing that a support exists is sufficient*.

AC-4 (Mohr and Henderson, 1986) is the only algorithm that confirms the existence of a support by not identifying it throughout search. However, it stores all supports for each value in auxiliary

data structures. Its inefficiency lies in its space complexity $\mathcal{O}(e\,d^2)$ and the necessity of maintaining huge data structures during search.

Coarse-grained algorithms such as AC-3 (Mackworth, 1977), AC-3.1/2001 (Bessiere et al., 2005) repeatedly revise the domains in order to remove unsupported values. However, in many revisions for difficult problems some or all values successfully find (identify) some support. This may result in long sequences of support checks. If we can guarantee the existence of *some* support for a given value without identifying it then a considerable amount of work in terms of checks and time can be saved.

In this paper, we shall show that evaluating the constraints before search can bring forth interesting knowledge. This knowledge can be captured in the form of *weights* and can be utilized effectively to reduce the cost of support inferencing in the coarse-grained arc consistency algorithms. In particular, we shall introduce the notions of a *Support Condition* (SC) and a *Revision Condition* (RC). If an SC holds, then it guarantees that a value has *some* support. It infers the existence of a support *without storing and maintaining support values*. The SCs help to avoid many (but not all) sequences of support checks. If an RC holds, then it guarantees that the SC holds for *each* value in a given domain which in turn guarantees that all values have *some* support. The RCs help to avoid many (but not all) ineffective revisions and much queue maintenance.

Both, the SC and RC are easy to implement. They have a limited overhead and a worst-case space requirement of $\mathcal{O}(e\,d)$, where e is the number of constraints and d is the maximum domain size. Results show that employing the SC and the RC in the coarse-grained algorithms significantly reduces support checks, ineffective revisions and solution time.

The remainder of this paper is as follows. Section 2 describes some relevant definitions and terminologies which are used throughout the paper. Sections 3 and 4 introduce the notions of the support and the revision conditions respectively. Section 5 presents an algorithm to extract information from constraints in the form of weights which are then used by the SC and RC for support inferencing. Section 6 shows the integration of the SC and RC in AC-3 and discusses related issues. Section 7 presents experimental results to show the efficiency of these conditions. Finally, conclusions are presented in Section 8.

2. Background

A constraint satisfaction problem is defined as a set $\mathcal{X}$ of n variables, a non-empty domain $D(x)$ for each variable $x \in \mathcal{X}$ and a set of e constraints defined among subsets of variables of $\mathcal{X}$. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of the domains $D(x)$ and $D(y)$, which specifies the allowed pairs of the values. The *tightness* of the constraint C_{xy} is defined as $p_t = 1 - |\,C_{xy}\,|/|\,D(x) \times D(y)\,|$.

Graphs theory plays a central role in capturing the structure of a constraint satisfaction problem and its solution process. The nodes of the constraint graph correspond to variables of the problem and the edges correspond to constraints. The *density* of a binary constraint graph is defined as $p_d = 2\,e/(n^2 - n)$. The *degree* of a variable is the number of constraints involving that variable. We use deg_{max} to denote the maximum degree of the variables. With each binary constraint C_{xy}, we associate two arcs (x, y) and (y, 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 shall use G to denote the directed constraint graph of the input CSP.

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* (also known as consistency or constraint check) is a test to find if two values support each other.

Definition 1 (Arc Consistency) *An arc (x, y) in the directed constraint graph of a* CSP *is arc-consistent if and only if every value $a \in D(x)$ has some support $b \in D(y)$. A* CSP *is arc-consistent if and only if every arc in its directed constraint graph is arc-consistent.*

Arc consistency ensures that any value in the domain of any variable has a support in the domain of any other selected variable. We call the domain of x after making the input CSP arc consistent for the first time the *first arc-consistent* domain of that variable. For the remainder of this paper for any variable x, we use $D_{ac}(x)$ for the first arc-consistent domain of x, and $D(x)$ for the current domain of x.

A *multi-set* is an unordered collection of elements from a universe. The elements in a multi-set are not necessarily distinct. For example, $\{1, 1, 1, 3\}$ as a set is actually equal to $\{1, 3\}$. However, as a multi-set, $\{\!\{1, 1, 1, 3\}\!\}$ is not simplifiable further. To avoid confusion, we use the notation $-_m$ to denote the asymmetric difference on multi-sets. For example, $\{\!\{1, 1, 1, 3\}\!\} -_m \{\!\{1, 3\}\!\}$ gives $\{\!\{1, 1\}\!\}$.

Definition 2 *An integer partition of a positive integer n is a multiset $p = \{\!\{\lambda_1, \lambda_2, \cdots, \lambda_m\}\!\}$ such that each λ_i is a positive integer and $sum(p) = \Sigma_{i=1}^{m} \lambda_i = n$. The λ_i are called the parts or elements of the partition.*

For example, the seven distinct integer partitions of 5 are $\{\!\{5\}\!\}$, $\{\!\{4, 1\}\!\}$, $\{\!\{3, 2\}\!\}$, $\{\!\{3, 1, 1\}\!\}$, $\{\!\{2, 2, 1\}\!\}$, $\{\!\{2, 1, 1, 1\}\!\}$ and $\{\!\{1, 1, 1, 1, 1\}\!\}$.

3. A Support Condition

In this section, we shall introduce the notion of a *support condition*. We shall first introduce its simple version which shall then facilitate us to present its generalized version.

3.1 A Simple Version of the Support Condition

Let C_{xy} be the constraint between x and y. Let $a \in D(x)$ be denoted as (x, a). Let *scount*$[\, x, y, a \,]$ be the number of supports of (x, a) in $D_{ac}(y)$. Let $R(y) = D_{ac}(y) \setminus D(y)$ be the set of the removed values from the first arc-consistent domain of y. Consequently, $|\, R(y) \,|$ is an upper bound on the number of supports of (x, a) removed from $D_{ac}(y)$. Therefore, if the following condition is true then (x, a) is supported by y:

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

We call the condition in Equation (1) (a simple version of) the support condition. Independent work by Boussemart et al. (2004) has also proposed an equivalent condition.

Let us illustrate the use of Equation (1). Consider the constraint C_{xy} as shown in Figure 1 (and for now ignore $w[x, y, a]$). For each value $a_i \in D(x)$, $1 \leq i \leq 4$, *scount*$[x, y, a_i] = 2$. Assume that exactly one value, say b_2, is removed from $D(y)$, that means $|R(y)| = 1$. We know that each value of $D(x)$ was supported by two values of $D_{ac}(y)$. Since only one value is removed from $D_{ac}(y)$, each value of $D(x)$ still has at least one support in $D(y)$. This is exactly what Equation (1) implies. If it holds then a support is guaranteed to exist and there is no need to identify a support.

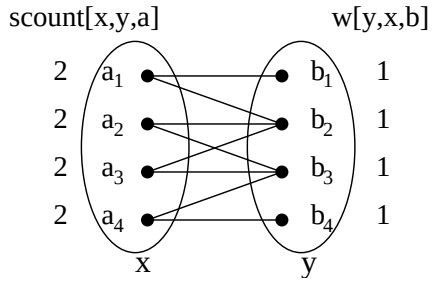

Figure 1: An example to illustrate the use of the simple version of support condition for support inferencing.

3.2 A Generalized Version of the Support Condition

Let $w[\,y, x, b\,]$ be some non-zero positive integer *weight* associated with $b \in D(y)$ with respect to x. The simple version of the SC shown in Equation (1) inherently assumes that $w[\,y, x, b\,] = 1$ for each value $b \in D(y)$ with respect to x.

Consider again Figure 1, and assume explicitly that $w[y, x, b] = 1$ for each value $b \in D(y)$ with respect to x. Notice that the support count of each value a_i is equal to the sum of the weights of its supports. For example, the support count of a_1 is equal to 2, the sum of the weights of b_1 and b_2. We can say that when the weight assigned to each value is 1,

$$\textit{scount}[\,x, y, a\,] = \sum_{b \,\in\, D_{ac}(y) \,\wedge\, (a,b) \,\in\, C_{xy}} w[\,y, x, b\,].$$

Similarly, we can say that when the weight assigned to each value is 1, the number of removed values are equal to the sum of the weights of the removed values, that is

$$|\,R(y)\,| = \sum_{b \,\in\, R(y)} w[\,y, x, b\,].$$

Thus, when the weight associated with each arc-value pair is 1, the Equation (1) can also be represented as follows:

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

We call the left hand side of Equation (2) the *cumulative weight* of (x, a) with respect to y, and denote it as $cw[\,x, y, a\,]$. It is equal to the sum of the weights of the values in $D_{ac}(y)$ supporting (x, a). We call the right hand side of Equation (2) the *removed weight* of y with respect to x, and denote it as $rw[\,y, x\,]$. It is equal to the sum of the weights of the values removed from $D_{ac}(y)$ with respect to x. Thus, Equation (2) can also be represented as follows:

$$cw[\,x, y, a\,] > rw[\,y, x\,] \tag{3}$$

When the weight associated with each arc-value pair is 1, the cumulative weight of (x, a) with respect to y is equal to the number of supports of (x, a) in $D_{ac}(y)$, the removed weight of y with

respect to x is equal to the number of values removed from $D_{ac}(y)$, and Equation (1) is a simple case of Equation (3). Note that if other non-zero positive integer weights are used and Equation (3) holds, then also a support is guaranteed. We call the condition in Equation (3) the generalized version of the *Support Condition* (SC).

Proposition 3 *Let W_s be the multi-set of the weights of the values in $D_{ac}(y)$ supporting $a \in D(x)$. Let W_r be the multi-set of the weights of the values in $R(y)$. If $W_s -_m W_r \neq \emptyset$ then $a \in D(x)$ has some support in $D(y)$.*

Consider a constraint C_{xy}, $a \in D(x)$ and $D_{ac}(y) = \{b_1, b_2, b_3, b_4, b_5\}$, wherein, the weight of b_1 is 1, b_2 is 2, b_3 is 1, b_4 is 2 and b_5 is 1. Also, assume that the set of supports of (x, a) in $D_{ac}(y)$ is $\{b_1, b_2, b_3\}$ and $R(y) = \{b_2, b_3, b_4\}$. It then follows that $W_s = \{\!\{1, 2, 1\}\!\}$ and $W_r = \{\!\{2, 1, 2\}\!\}$. Note that $W_s -_m W_r \neq \emptyset$, which means that some element of W_s is not in W_r. Here that element is 1. Since there are two 1s in W_s and only single 1 in W_r, it implies that either b_1 or b_3 is not in $R(y)$. Therefore, one can conclude that at least one of them is in $D(y)$ and $a \in D(x)$ has at least one support in $D(y)$.

Theorem 4 *Let cw be the cumulative weight of $a \in D(x)$ with respect to y. Let rw be the removed weight of y with respect to x. If $cw > rw$ then $a \in D(x)$ is supported by some value of $D(y)$.*

Proof Let C denote the set of multi-sets of all integer partitions of cw. Let R denote the set of multi-sets of all integer partitions of rw. If $cw > rw$ then it follows that $\forall c \in C, \forall r \in R$, $c -_m r \neq \emptyset$. Let W_s denote the multi-set of the weights of the values in $D_{ac}(y)$ supporting $a \in D(x)$ and W_r denote the multi-set of the weights of the removed values. From the definition of cumulative weight, we have $cw = sum(W_s)$ and from the definition of removed weight, we have $rw = sum(W_r)$. Therefore, $W_s \in C$, $W_r \in R$ and $W_s -_m W_r \neq \emptyset$. It now follows from Proposition 3 that $a \in D(x)$ has some support in $D(y)$. ■

3.3 Impact of using Different Weights

We shall now illustrate the impact of using different weights on the efficiency of the SC. In Figure 2, Case 1, the weight associated with each value of $D(y)$ is 1, whereas, in Case 2, it is its own support count. Let us study the behavior of the SC while revising $D(x)$ against $D(y)$, when one or more values are removed from $D(y)$.

If any single value is removed from the domain of y, then the SC will hold for each value of $D(x)$ in both the cases. Hence, there is no need to seek a support. For example, if b_1 is removed, then $rw[\, y, x\,] = 1$ for both the cases. A support is guaranteed for any value of $a \in D(x)$ in $D(y)$, since $cw[\, x, y, a\,]$ is greater than $rw[\, y, x\,]$.

Now, if b_4 is also removed, then $rw[\, y, x\,]$ becomes 2 for both the cases. Although b_1 and b_4 are removed, all the values of $D(x)$ will still be supported by some value of $D(y)$. However, the SC will fail for each value of $D(x)$ for Case 1, since the cumulative weight of any value of $D(x)$ is not greater than the removed weight of y with respect to x. But, in Case 2, the cumulative weight of each value of $D(x)$ is greater than 2. Thus, the SC will infer the support existence for each value of $D(x)$ without requiring to seek a support.

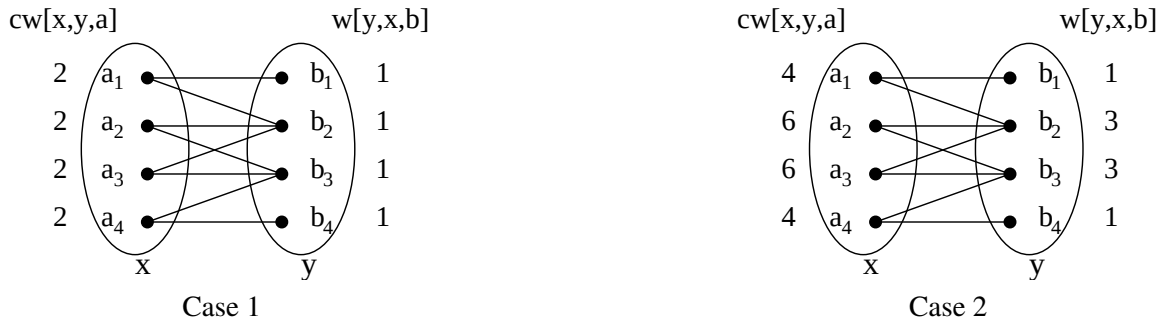

Figure 2: Weights of the values: in Case 1 the weight associated with each b_i is 1 and in Case 2 the weight associated with each b_i is its own support count with respect to x. The cumulative weight of each $a_i \in D(x)$ is computed for each case accordingly.

In another scenario, if only b_1 and b_2 are removed, then in Case 1, $rw[\,y, x\,] = 2$ and in Case 2, $rw[\,y, x\,] = 4$. The SC will again fail for each value in Case 1. However, in Case 2, the SC will hold for at least a_2 and a_3, since their cumulative weights are greater than 4. This shows that the efficiency of the SC may depend on the weights assigned to the values.

3.4 How to assign weights?

A natural question arises: is there some way of assigning weights to arc-value pairs so that support inferencing using SC can be increased. *One possible way is to assign a weight to a value of a domain such that it is greater than the aggregate of the weights of all the values which will be removed from the domain before that value.*

Let us consider the same constraint C_{xy} as illustrated in Figure 1. We shall use $b_i <_r b_j$ to indicate that b_i is removed before b_j. Assume that the order in which the values are removed from the domain of y is $b_1 <_r b_2 <_r b_3 <_r b_4$ that is, first b_1 is removed, then b_2 and so on. For now, we shall omit the use of variables x and y in $w[x, y, b]$ for the comfort of representation.

Following the idea mentioned before, the weight of b_1 should be the lowest, since b_1 will be deleted first. The weight of b_2 should be greater than the weight of b_1, *i.e.* $w[b_2] > w[b_1]$, since b_1 will be removed before b_2. Similarly, $w[b_3] > w[b_2] + w[b_1]$ and $w[b_4] > w[b_3] + w[b_2] + w[b_1]$. One way of assigning the weights to the values, while satisfying these conditions is possible with the help of the following equation:

$$w[\,y, x, b_j\,] = 1 + \sum_{b_i \in D_{ac}(y)\ \wedge\ b_i <_r b_j} w[y, x, b_i]. \tag{4}$$

Figure 3 depicts the weights assigned to the values of the domain of y by using Equation (4). The cumulative weights of a_1, a_2, a_3 and a_4, therefore, become 3, 6, 6, and 12 respectively. Now, let us study the consequences of the removal of the values from $D_{ac}(y)$ in the increasing order of their subscripts.

When only b_1 is removed, there is no need to seek a support for any value of $D(x)$, since the SC succeeds for each value of $D(x)$. When the two values b_1 and b_2 are deleted, $rw[\,y, x\,]$ becomes 3. Unlike Case 2 of Figure 2, where the SC was unsuccessful for a_1 and a_4 here it fails only for

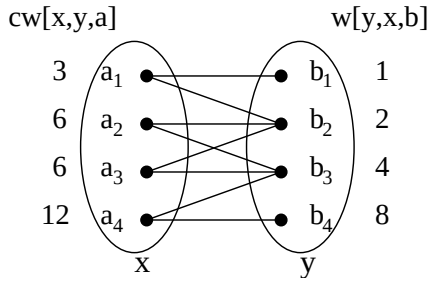

Figure 3: The weight for each $b_i \in D(y)$ is computed using Equation (4) based on which the cumulative weight for each a_i is evaluated.

a_1. More importantly, the SC fails for a_1, when all its supports are deleted. When b_1, b_2 and b_3 are deleted, the removed weight of y becomes 7 and the SC fails for a_1, a_2 and a_3. Again, notice that the SC fails for a given value only when the value has lost all its supports and is bound to get removed.

In summary, if the order in which the values are removed is known beforehand, then using Equation (4) to compute the weights empowers SC to infer the support existence for a given arc-value pair as long as the value has at least one support. It fails only when the value loses all its supports. So, whenever the SC fails, the corresponding value can be removed without verifying the absence of a support. This suggests that not a single support check is required if the SC is used while revising a domain. However, knowing the order in which the values are removed from the domains in advance is almost impossible. Yet, this can be used as a guiding factor to assign the weights to the values.

A general strategy should be: the value with a less likelihood of staying in the domain for a longer time should be assigned a lower weight and vice-versa. One way of measuring this likelihood is to use support counts as weights. An illustration of this is shown in Case 2 of Figure 2. A value having a lower support count is more likely to be removed before a value having a higher support count. Not that a variable is usually constrained by more than one variable in a given CSP. In such a situation, a value may have more supports in one variable's domain and fewer in another. Assigning just support counts as weights does not take this into consideration. Therefore, instead of support counts, the sum of the support counts can also be used as weights. For example, if there is a value $a \in D(x)$ and if x is constrained with y and z then the sum of the support counts of (x, a) with respect to y and z can be used as its weight with respect to both y and z. We will see further on (in Section 7) that using support counts and sums of support counts as weights allows the SC to save more checks.

4. A Revision Condition

The support check is the core operation performed by arc consistency algorithms. Most of the improvements suggested so far to reduce the number of support checks are done at a fine level of granularity, that is at the level of arc-value pair. For example, validity checks in AC-3.1, checking the status of the support counters in AC-4, etc. We call all these tests *auxiliary support checks* (ASCs). When support checks are not too expensive then auxiliary support checks can be an overhead and reducing the support checks *alone* does not always help in reducing the solution time. In

this section, we shall propose a *coarser check* at arc level to avoid a complete revision which will not only save support checks but also auxiliary support checks.

For a given arc, (x, y), if the least cumulative weight of the values of $D(x)$ with respect to y is greater than the removed weight of y with respect to x, then it follows that the cumulative weight of all the values of $D(x)$ are greater than the removed weight from y. Therefore, if the following condition holds, then each value in $D(x)$ is guaranteed to have a support in $D(y)$:

$$\min \{cw[\, x, y, a\,] : a \in D(x)\} > rw[\, y, x\,]\,. \tag{5}$$

The drawback of using Equation (5) is that for any given arc, (x, y), it requires the least cumulative weight of the values $D(x)$ with respect to y, which may change during search as values are pruned. Therefore, it may be necessary to recompute and maintain the least cumulative weight of $D(x)$ with respect to y for each arc (x, y) during search, which can be an overhead. Nevertheless, there is a possibility of reducing this overhead by weakening the condition presented in Equation (5) at the cost of performing few more ineffective revisions.

We define the *cumulative weight of an arc*, (x, y), as less than or equal to the least cumulative weight of the values of $D(x)$ with respect to y and denote it as $cw[x, y]$, which can be expressed as follows:

$$cw[x, y] \leq \min \{cw[\, x, y, a\,] : a \in D(x)\}\,. \tag{6}$$

If the cumulative weight of an arc, (x, y), is greater than the removed weight from y, then it follows that the cumulative weights of all the values of $D(x)$ are also greater than the removed weight from y. Therefore, if the following condition holds then each value in $D(x)$ is guaranteed to have a support in $D(y)$:

$$cw[\, x, y\,] > rw[\, y, x\,]\,. \tag{7}$$

We call this condition a Revision Condition (RC). If the RC holds for a given arc, (x, y), then it guarantees that *all* values in $D(x)$ have some support in $D(y)$ and a complete revision can be avoided. In the following sections, we shall present the *dynamic*, *partially dynamic* and *static* versions of the RC. The difference lies in the frequency by which the cumulative weights of the arcs are updated during search.

4.1 The Dynamic Revision Condition

In the Dynamic Revision Condition (DRC), the cumulative weight of an arc, (x, y), is equivalent to the least cumulative weight of the values of $D(x)$ with respect to y. This can be expressed as follows:

$$cw[x, y] = \min \{cw[\, x, y, a\,] | a \in D(x)\}\,.$$

Note that if the revision of $D(x)$ against some variable's domain removes even one value, then the least cumulative weight of $D(x)$ with respect to all its neighbors y may change. Therefore, for the DRC, there is a need to recompute the least cumulative weight of $D(x)$ with respect to each y with which x is constrained after every effective revision of $D(x)$.

4.2 The Partially Dynamic Revision Condition

Computing the least cumulative weight of $D(x)$ with respect to each y with which x is constrained after every effective revision of $D(x)$ can be an overhead. One alternative is to compute the least

cumulative weight of $D(x)$ with respect to y only when $D(x)$ is revised against $D(y)$. This can be done cheaply and may avoid the ineffective revision of $D(x)$ against $D(y)$ in future. Here, the cumulative weight of an arc, (x, y), is less than or equal to the least cumulative weight of the values of $D(x)$ with respect to y, which can be expressed as follows:

$$cw[x, y] \leq \min \{cw[\, x, y, a\,] | a \in D(x)\}\,.$$

We call it a Partially Dynamic Revision Condition (PDRC) because the cumulative weights of all the affected arcs are not updated after every effective revision. The disadvantage is that all possible ineffective revisions cannot be saved which can be saved by the dynamic RC.

4.3 The Static Revision Condition

If the cumulative weight of an arc, (x, y), is the least cumulative weight of the values of the first arc consistent domain of x then it remains static throughout the search. This can be expressed as follows:

$$cw[x, y] = \min \{cw[\, x, y, a\,] | a \in D_{ac}(x)\}\,.$$

We call it a Static Revision Condition (SRC). In this setting, once the cumulative weights of the arcs are initialized, they are never updated. This may not reduce as many ineffective revisions as can be reduced by the dynamic or the partially dynamic revision conditions.

Independent work by Boussemart et al. (2004) has proposed a condition similar to the SRC. However, their proposed condition is a special case of the SRC, where the weight associated to each arc-value pair is 1.

5. Computation of the Cumulative Weights

In order to use the SC and the RC a non-zero positive integer weight should be assigned to each arc-value pair and the cumulative weight should be computed for each arc-value pair and arc. This can be done either before or after arc consistency preprocessing. However, there are at least three advantages of doing it after arc consistency preprocessing, which are as follows:

- Occasionally, enforcing arc consistency alone can deliver a solution without requiring further search for some problems, in which case the work required to compute the cumulative weights can be saved.
- For some over-constrained problems, applying arc consistency can prove the inconsistency of the constraint network without search. In such cases also, the computation of the cumulative weights is unnecessary.
- Computing the cumulative weights after arc consistency preprocessing may help the SC and the RC in reducing more checks and ineffective revisions respectively.

The procedure *assignWeights* is used to assign some weight to *weight*$[\, x, y, a\,]$, for each arc-value pair, involving the arc (x, y) and the value a in $D_{ac}(x)$. The procedure *computeCumulativeWeights* is used to compute the cumulative weights for each arc-value pair and each arc. The implementations of these two procedures are left open. In practice, they are integrated to effectively compute the cumulative weights. However, for the sake of simplicity and easy understanding, we shall treat them separately.

The implementation of *assignWeights* varies with respect to the criterion used to assign a weight to each arc-value pair. Assigning the weight 1 to each arc-value pair is straightforward. But, when the weight assigned to each arc-value pair is its own support count, effort has to be made to compute the support counts. There may exist many different criteria to assign the weights. It is out of the scope of this paper to investigate them and present their corresponding algorithms.

The efficient implementation of *computeCumulativeWeights* depends upon the way constraints are represented. Here efficient implementation refers to minimizing the number of support checks. The algorithm presented in Figure 4 is general in the sense that it does not depend upon the specification of constraints. It is certainly not competitive with an algorithm which can be written for specific types of constraints. For example, it is possible to exploit the semantics of the intensional constraints.

The algorithm presented in Figure 4 assumes that some reasonable weight is assigned to each arc-value pair. It exploits the bidirectional property of constraints to halve the number of support checks (line 13). The algorithm also assumes that an optimal coarse-grained algorithm such as AC-3.1 is used to make the problem arc consistent before computing the cumulative weights. Note that AC-3.1's *last*$[\,x, y, a\,]$ data structure stores last known support of (x, a) with respect to y which is also the first support of (x, a) in $D_{ac}(y)$. The idea is to explore the values in $D_{ac}(y)$ that are after the first support of $a \in D(x)$ to find the remaining supports (line 10). This may save some support checks. The worst-case time complexity of the algorithm is $\mathcal{O}(e\,d^2)$.

```
procedure computeCumulativeWeights ( )
  begin
  1: for each C_xy ∈ C do
  2:    for each a ∈ D_ac(x) do
  3:       cw[x, y, a] := 0
  4:    for each b ∈ D_ac(y) do
  5:       cw[y, x, b] := 0
  6:    for each a ∈ D_ac(x) do
  7:       b' := last[x, y, a]
  8:       cw[x, y, a] := weight[y, x, b']
  9:       cw[y, x, b'] := cw[y, x, b'] + weight[x, y, a]
  10:      for each b ∈ D_ac(y) such that b > last[x, y, a] do
  11:         if b supports a then
  12:            cw[x, y, a] := cw[x, y, a] + weight[y, x, b]
  13:            cw[y, x, b] := cw[y, x, b] + weight[x, y, a]
  14:   cw[x, y] := min{cw[x, y, a] | a ∈ D_ac(x)}
  15:   cw[y, x] := min{cw[y, x, b] | b ∈ D_ac(y)}
  end
```

Figure 4: An algorithm to compute cumulative weights.

6. Integrating SC and RC in AC-3

In this section, we shall be concerned with the integration of the SC and the RC in AC-3.

If the RC holds, then it can be exploited either *after* selecting an arc, (x, y), from the queue for the next revision or *while* adding the arcs 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 RC by tightening the condition for adding the arcs to the queue: arcs should be added only if the RC does not hold. This is depicted in line 8 of Figure 5. With this implementation the advantages of using the RC are threefold:

- It reduces the number of arcs that have to be added to the queue, which reduces the overhead of queue management and saves time.
- It reduces the number of arcs that are present in the queue. This speeds up the selection of the optimal arc.
- The number of ineffective revisions are reduced. This in turn reduces the number of support checks and in case of an optimal coarse-grained algorithm, it also reduces auxiliary support checks.

```
Function AC-3: Boolean;
begin
1:    Q := G
2:    while Q not empty do
3:        select and remove any arc (x, y) from Q
4:        revise(x, y, change_x)
5:        if D(x) = ∅ then
6:            return False
7:        if change_x then
8:            Q := Q ∪ { (y', x) | y' is a neighbour of x ∧ y' ≠ y∧ cw[y', x] ≤ rw[x, y'] }
9:    return True;
end
```

Figure 5: AC-3 equipped with RC.

Pseudo-code for the $revise$ function of AC-3 is depicted in Figure 6. It is different from its original version (Mackworth, 1977) because it uses the SC to avoid a series of checks for which it can be known in advance that it will eventually lead to a support (lines 3-4). If $a \in D(x)$ fails to find a support in $D(y)$, then it is removed from $D(x)$ and the removed weight of x with respect to each y' with which x is constrained is updated (lines 7–8).

```
Function revise(in x, in y, out change_x)
begin
1:  change_x := False
2:  for each a ∈ D(x) do
3:      if cw[x, y, a] > rw[y, x] then
4:          continue /* a is supported */
5:      else if ∄b ∈ D(y) such that b supports a then begin
6:          D(x) := D(x) \ { a }
7:          for each y' such that (y', x) ∈ G do
8:              rw[y', x] := rw[y', x] − weight[x, y', a]
9:          change_x := True
end
```

Figure 6: Algorithm $revise$ of AC-3 equipped with SC.

Note that the cumulative weights associated with arc-value pairs remain static during search. Hence, there is no overhead of maintaining them. However, the cumulative weights associated with arcs may change depending upon the version of the RC used in AC-3. The worst-case time complexity of updating the cumulative weight of a single arc is $\mathcal{O}(d)$.

The integration of the RC in AC-3 is presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. When the SRC is used, there is no overhead of maintaining the cumulative weights of the arcs, since they are never updated. When the PDRC is used, the cumulative weight of an arc needs to be updated when that arc is considered for

the next revision. When the DRC is used, the cumulative weights of at most deg_{max} arcs may need to be updated after an effective revision.

Note that a 2-dimensional array *rw* is used to store the removed weights. In the actual implementation, *rw* does not always need to be a 2-dimensional array. If the weight assigned to a given value is different with respect to different arcs then the removed weights of the affected arcs are updated with the weight of the removed value corresponding to those arcs respectively. In such a case, the 2-dimensional array is required. However, when the weight assigned to a value is the same with respect to different arcs, removing a value will update the removed weight of the affected arcs by the same weight. Therefore, in such a case, one dimensional-array is sufficient.

The space complexity of storing the cumulative weights of the arc-value pairs is $\mathcal{O}(e\,d)$. Initially, the space complexity of storing the removed weights and the cumulative weights of the arcs is $\mathcal{O}(e)$. But depending upon the version of the RC, the worst-case space complexity may increase to $\mathcal{O}(e\,d)$ during search. The overall space complexity of using the SC in conjunction with the RC is $\mathcal{O}(e\,d)$.

7. Experimental Results

In this section, we shall present some empirical results to demonstrate the practical efficiency of the *support condition* and the *revision condition*.

We use the SC and RC in the arc consistency components of MAC-3 and MAC-3.1. The arc consistency components of both the algorithms were equipped with *comp* (Van Dongen, 2004) as a revision ordering heuristic. During search all MACs visited the same nodes in the search tree. They were equipped with a *dom/deg* (Bessière and Régin, 1996) variable ordering heuristic.

We used three different measures of assigning the weights to arc-value pairs: w_1 denotes that the weight associated with each arc-value pair is 1, w_{sc} denotes that the weight associated with each arc-value pair is its own support count and $w_{\Sigma sc}$ denotes that the weight associated with each arc-value pair is the sum of the support counts of the value with respect to each constraint it is involved in.

The experiments were performed on random problems, RLFAP problems and quasi-group problems with holes (QWH). They are described in the following sections with their corresponding results. Performance is measured in terms of the number of support checks, the solution time (in seconds) and the number of revisions. All algorithms were written in C. The experiments were conducted on a PC Pentium III (2.266 GHz processor and 256 MB RAM).

7.1 Random Problems

We experimented with the model B random problems. In this model, a random CSP instance is characterized by $\langle\, n, d, p_d, p_t \,\rangle$ where n is the number of variables, d is the uniform domain size, p_d is the density of the constraint graph and p_t is the uniform tightness of each constraint. We generated instances for $n = 50$, $d = 10$, $(p_d, p_t) \in \{(0.20, 0.36),\ (0.40, 0.20),\ (0.60, 0.14),\ (0.80, 0.13),\ (1.00, 0.12)\}$. For each combination of $\langle n, d, p_d, p_t \rangle$, 50 instances were generated. Tables 1–3 present mean results for the problem classes $\langle 50, 10, 0.20, 0.36 \rangle$, $\langle 50, 10, 0.60, 0.14 \rangle$, and $\langle 50, 10, 1.00, 0.12 \rangle$ respectively. The problem parameters were chosen deliberately to show the efficiency of the SC and the RC on the problems which are located in the phase transition (Cheeseman et al., 1991). For random problems support checks were implemented as cheap lookup array operations.

Condition	Weight	MAC-3 Checks	MAC-3 Time	Revisions	MAC-3.1 Checks	MAC-3.1 Time
-	-	4,441,840	0.350	768,330	1,393,532	0.403
SC	w_1	2,586,069	0.330	768,330	1,277,170	0.368
SC + DRC	w_1	2,586,069	0.363	641,941	1,277,170	0.436
SC + PDRC	w_1	2,586,069	0.339	678,956	1,277,170	0.388
SC + SRC	w_1	2,586,069	0.315	685,822	1,277,170	0.354
SC	w_{sc}	2,096,077	0.395	768,330	1,137,447	0.448
SC + SRC	w_{sc}	2,096,077	0.370	648,807	1,137,447	0.421
SC	$w_{\Sigma sc}$	2,168,242	0.329	768,330	1,176,443	0.372
SC + SRC	$w_{\Sigma sc}$	2,168,242	0.303	654,978	1,176,443	0.356

Table 1: Mean results for the random problems $\langle 50, 10, 0.20, 0.36 \rangle$.

Condition	Weight	MAC-3 Checks	MAC-3 Time	Revisions	MAC-3.1 Checks	MAC-3.1 Time
-	-	1,522,785,770	161.329	389,148,996	478,439,578	186.139
SC	w_1	414,748,936	165.230	389,148,996	279,726,796	182.498
SC + DRC	w_1	414,748,936	138.576	205,415,241	279,726,796	196.237
SC + PDRC	w_1	414,748,936	141.799	263,147,033	279,726,796	158.655
SC + SRC	w_1	414,748,936	137.367	279,471,217	279,726,796	152.998
SC	w_{sc}	295,815,188	198.638	389,148,996	221,013,167	219.326
SC + SRC	w_{sc}	295,815,188	153.456	227,939,382	221,013,167	171.796
SC	$w_{\Sigma sc}$	309,733,345	162.530	389,148,996	227,008,534	177.562
SC + SRC	$w_{\Sigma sc}$	309,733,345	118.859	238,484,341	227,008,534	134.982

Table 2: Mean results for the random problems $\langle 50, 10, 0.60, 0.14 \rangle$.

Tables 1–3 show that the SC reduces the number of support checks and the RC reduces the number of revisions. For instance, for the random problem class $\langle 50, 10, 1.00, 0.12 \rangle$ in Table 3, the support checks required by MAC-3 and MAC-3.1 are reduced by at least 90% and 72% respectively by using only SC. Nonetheless, the solution time is not reduced much. This shows that performing auxiliary support checks to reduce support checks is not a great help in reducing the overall solution time, especially when the support checks are cheap. As expected, the DRC saves more ineffective revisions than the PDRC, which in turn saves more ineffective revisions than the SRC. On the contrary, the SRC is more efficient in terms of time than the PDRC, which in turn is more efficient than the DRC. This is because there is an overhead to update the cumulative weights of arcs for the DRC and the PDRC. The results for the algorithms when they are equipped with the DRC or the PDRC are shown only with respect to w_1.

Note that when the weights are assigned using w_1 or $w_{\Sigma sc}$ criteria, removing a single value from a given variable's domain requires only $\mathcal{O}(1)$ operation to update the removed weight of the variable with respect to all its neighbors with which it is constrained. Thus, 1-dimensional array *rw* is sufficient. However, when w_{sc} is used, a 2-dimensional array is required for *rw*, since the weight associated with a given value may be different with respect to different arcs. Therefore, $\mathcal{O}(deg_{max})$ operations are performed in the worst-case after every deletion to update the removed weights, which is an overhead. Therefore, the SC + SRC spends more time with respect to w_{sc} than they spend with respect to w_1 or $w_{\Sigma sc}$.

Condition	Weight	MAC-3 Checks	MAC-3 Time	Revisions	MAC-3.1 Checks	MAC-3.1 Time
-	-	194,469,206	23.244	41,957,598	54,027,225	24.473
SC	w_1	26,592,399	20.371	41,957,598	18,832,724	22.423
SC + DRC	w_1	26,592,399	15.072	13,482,255	18,832,724	23.143
SC + PDRC	w_1	26,592,399	13.737	18,789,129	18,832,724	15.341
SC + SRC	w_1	26,592,399	13.245	20,007,585	18,832,724	14.965
SC	w_{sc}	19,153,698	25.741	41,957,598	15,080,588	27.951
SC + SRC	w_{sc}	19,153,698	16.457	14,110,869	15,080,588	18.023
SC	$w_{\Sigma sc}$	20,870,288	19.545	41,957,598	16,074,448	21.677
SC + SRC	$w_{\Sigma sc}$	20,870,288	10.048	15,627,124	16,074,448	12.489

Table 3: Mean results for the random problems $\langle 50, 10, 1.00, 0.12 \rangle$.

7.2 Radio Link Frequency Assignment Problem

A Radio Link frequency Assignment Problem (RLFAP) is to assign frequencies to a set of radio links defined between pairs of sites in order to avoid interferences (Cabon et al., 1999). Tables 4 and 5 correspond to the result of RLFAP#11, when solved with MAC-3 and MAC-3.1respectively.

Condition	Weight	initial Checks	initial Time	Search Checks	Search Time	Total Checks	Total Time	Revisions
-	-	971,893	0.018	41,547,613	1.965	42,519,506	1.982	1,602,603
SC	w_1	7,010,181	0.097	21,419,292	1.342	28,429,473	1.439	1,602,603
SC + DRC	w_1	7,010,181	0.103	21,419,292	1.173	28,429,473	1.277	750,731
SC + PDRC	w_1	7,010,181	0.096	21,419,292	1.003	28,429,473	1.099	752,277
SC + SRC	w_1	7,010,181	0.102	21,419,292	0.949	28,429,473	1.051	752,312
SC	w_{sc}	7,010,181	0.163	21,268,715	1.947	28,278,896	2.110	1,602,603
SC + SRC	w_{sc}	7,010,181	0.162	21,268,715	1.468	28,278,896	1.630	746,169
SC	$w_{\Sigma sc}$	14,022,597	0.355	21,036,914	1.395	35,059,511	1.750	1,602,603
SC + SRC	$w_{\Sigma sc}$	14,022,597	0.358	21,036,914	0.992	35,059,511	1.350	746,190

Table 4: Results for RLFAP#11 when solved with MAC-3.

Condition	Weight	initial Checks	initial Time	Search Checks	Search Time	Total Checks	Total Time	Revisions
-	-	971,893	0.025	9,361,105	1.585	10,332,998	1.610	1,602,603
SC	w_1	7,010,181	0.098	8,136,873	1.487	15,147,054	1.585	1,602,603
SC + DRC	w_1	7,010,181	0.096	8,136,873	1.460	15,147,054	1.556	750,731
SC + PDRC	w_1	7,010,181	0.095	8,136,873	1.124	15,147,054	1.220	752,277
SC + SRC	w_1	7,010,181	0.098	8,136,873	1.059	15,147,054	1.157	752,312
SC	w_{sc}	7,010,181	0.162	8,201,259	2.146	15,211,440	2.308	1,602,603
SC + SRC	w_{sc}	7,010,181	0.166	8,201,259	1.662	15,211,440	1.828	746,169
SC	$w_{\Sigma sc}$	14,022,597	0.377	7,985,510	1.508	22,008,107	1.885	1,602,603
SC + SRC	$w_{\Sigma sc}$	14,022,597	0.381	7,985,510	1.100	22,008,107	1.482	746,190

Table 5: Results for RLFAP#11 when solved with MAC-3.1.

The column labeled as "initial" refer to the stage of making the problem arc-consistent before search and the work, if any, required to compute the cumulative weights. The overhead of computing the cumulative weights increases the time required to finish the initial stage by an order of magnitude which is between 5 and 20, depending upon the criterion used to assign weight. However, this initial investment of computing the cumulative weights pays off by avoiding at least 50% of the total revisions using any version of the RC. When the SC and the SRC are used together and w_1 is used to assign the weights, there is a 50% and 30% reduction in the solution time of MAC-3 and MAC-3.1 respectively.

7.3 Quasigroup Problem with Holes

Table 6 corresponds to the mean results of five instances of balanced Quasigroup problem With Holes (QWH) (Achlioptas et al., 2000) of order 20. The instances were used as benchmarks for the First International Constraint Satisfaction Solver Competition. The procedure of generating the instances is described in (Boussemart et al., 2005).

		MAC-3			MAC-3.1	
Condition	Weight	Checks	Time	Revisions	Checks	Time
-	-	236,971,905	65.763	91,198,921	89,924,612	72.062
SC	w_1	37,787,658	63.975	91,198,921	24,041,064	68.731
SC + DRC	w_1	37,787,658	47.793	28,701,213	24,041,064	70.260
SC + PDRC	w_1	37,787,658	42.423	38,953,813	24,041,064	45.692
SC + SRC	w_1	37,787,658	40.362	38,953,813	24,041,064	44.421
SC	w_{sc}	32,307,142	86.243	91,198,921	21,890,725	88.840
SC + SRC	w_{sc}	32,307,142	45.860	20,692,635	21,890,725	48.384
SC	$w_{\Sigma sc}$	33,046,110	63.662	91,198,921	22,250,939	66.465
SC + SRC	$w_{\Sigma sc}$	33,046,110	32.346	27,166,530	22,250,939	34.645

Table 6: Results for Quasigroup Problem with Holes of order 20.

When the SC is used in MAC-3 and MAC-3.1, the number of support checks are reduced by at least 84% and 73% respectively, which is significant. Yet, there is only a marginal saving in terms of solution time. It is interesting to note that MAC-3 with the SC requires fewer checks than the original version of MAC-3.1. Remember that MAC-3 uses a non-optimal algorithm AC-3 while MAC-3.1 uses an optimal algorithm AC-3.1. When the revision condition is used in the algorithms, the number of revisions are reduced by at least 50%, which results in saving significant amount of time. When the SC and the SRC are used together and $w_{\Sigma sc}$ is used as a criterion to compute and assign the weights, there is on average, a 50% reduction in the solution time of MAC-3 and MAC-3.1.

Any instance of QWH problem basically consists of anti-functional constraints. If a constraint C_{xy} is an anti-functional constraint, then for each value $a \in D(x)$ there is at most one value $b \in D(y)$ which is not a support of a. For such a constraint, when the SRC is used in any coarse-grained arc consistency algorithm with respect to w_1, $D(x)$ is never considered for the revision as long as $|D(y)| > 1$. Note that the algorithm AC-5 Van Hentenryck et al. (1992) uses a tailored procedure for anti-functional constraints. If it is used on the instances of QWH, then an arc, (x, y), is considered for the revision only when $|D(y)| = 1$.

An anti-functional constraint can sometimes also be a universal constraint, *i.e.* a constraint which holds for all possible pairs of values of given domains. When AC-5 encounters such a constraint, C_{xy}, it revises $D(x)$, when $|D(y)| = 1$, and this results in an ineffective revision. Such ineffective revisions can be avoided by using the revision condition. Moreover, tailored algorithms are limited to the problems for which they are designed. A slight change in the problem specification would render the algorithm inapplicable. For example, instead of conflicting with at the most one value, if just a single value conflicts with 2 values, then AC-5 cannot take the advantage of this knowledge. However, both the SC and the RC can take advantage and can save some ineffective propagation. The SC and the RC not only improve the generic filtering but also reduce the gap between the generic and the specific constraint propagation procedures, which is also one of the challenges of the constraint programming community.

8. Conclusion

In this paper, we showed that evaluating the constraints before search can provide interesting and useful knowledge which can be used to reduce the cost of support inferencing. We introduced the notions of the support condition and the revision condition. We further introduced the dynamic, partially dynamic and static versions of the revision condition. If the SC holds, then it guarantees the existence of some support without identifying it. The SCs avoid many sequences of support checks. If the RC holds, then it guarantees the existence of some support for each value in a given domain and avoids the corresponding revision. The RCs reduce the number of arcs that have to be added to the queue. Having fewer arcs in the queue improves the process of selection of the best arc from the queue. Furthermore fewer revisions are performed. In short, the SC and the RC improve the performance of generic arc consistency algorithms by reducing ineffective constraint propagation.

The efficiency of the SC and the RC in reducing the checks and revisions depends on the weights assigned to arc-value pairs. In general, a value with a less likelihood of staying in the domain for a longer time should be assigned a lower weight and vice-versa. Empirical results suggest that when support checks are cheap, reducing them by using auxiliary support checks such as the SCs, does not payoff a lot in terms of the solution time. However, the use of the RC saves time, since it not only reduces support checks but also auxiliary support checks and queue maintenance. The overhead of maintaining the cumulative weights of the arcs in the dynamic and the partially dynamic versions of the RC penalize the algorithms in terms of time when compared to the static version of the RC, where there is no such overhead.

Acknowledgments

I would like to thank Marc van Dongen and the anonymous reviewers for their useful comments which helped to improve this paper.

References

D. Achlioptas, C.P. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceedings of the Seventeenth AAAI/IAAI Conference*, pages 256–261, 2000.

C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ ?) on hard problems. In E.C. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 61–75. Springer, 1996.

C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithms. *Artificial Intelligence*, 165(2):165–185, 2005.

F. Boussemart, F. Hemery, and C. Lecoutre. Description and representation of the problems selected for the first international constraint satisfaction solver competition. In M.R.C. van Dongen, editor, *Proceedings of the Second International Workshop on Constraint Propagation and Implementation, Volume 2 Solver Competition*, pages 7–26, 2005.

F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Support inference for generic filtering. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, pages 721–725, Toronto, Canada, 2004.

B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Journal of Constraints*, 4:79–89, 1999.

P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of the twelfth International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 331–337, 1991.

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

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

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.

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

P. Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, 1992.