

# CSP Search Algorithms with Responsibility Sets and Kernels \*

Igor Razgon and Amnon Meisels  
Department of Computer Science,  
Ben-Gurion University of the Negev,  
Beer-Sheva, 84-105, Israel  
{irazgon,am}@cs.bgu.ac.il

**Abstract.** A CSP lookahead search algorithm, like FC or MAC, explores a search tree during its run. Every node of the search tree can be associated with a CSP created by the refined domains of unassigned variables. If the algorithm detects that the CSP associated with a node is insoluble, the node becomes a dead-end. A strategy of pruning "by analogy" states that the current node of the search tree can be discarded if the CSP associated with it is "more constrained" than a CSP associated with some dead-end node.

In this paper we present a method of pruning based on the above strategy. The information about the CSPs associated with dead-end nodes is kept in the structures called responsibility set and kernel. The method that uses these structures for pruning is termed Responsibility set, Kernel, Propagation - RKP. The resulting combined algorithms are FC-RKP and MAC-RKP. Under certain restrictions, FC-RKP is shown theoretically to simulate FC-CBJ. Experimental evaluation is presented demonstrating that MAC-RKP outperforms MAC-CBJ on random CSPs and on random graph coloring problems.

## 1. Introduction

CSP search algorithms use methods for pruning of the search space. Well-known pruning methods restrict the search space by achieving some level of local consistency and removing the 'locally-inconsistent' values (Prosser, 1993; Sabin and Freuder, 1994). However, there are problem instances where maintaining local consistency provides little help, for example a CSP with all variables connected by the inequality constraints. Such instances frequently appear as small parts of real-world problems. The difficulty of such instances inspired the CSP community to look for pruning methods based on other principles than maintaining of local consistency. The strategic direction that has been proposed is the development of pruning methods for special types of constraints. Methods of constraint propagation (Regin, 1994; Quim-

---

\* Partially supported by the Lynn and William Frankel Center for Computer Science and by the Paul Ivanier Center for Robotics

per et al., 2004) and of symmetry breaking (Puget, 2005; Focacci and Milano, 2001; Fahle et al., 2001) fall into this category.

One of the most popular symmetry breaking methods is Symmetry Breaking by Dominance Detection (SBDD) (Puget, 2005; Focacci and Milano, 2001; Fahle et al., 2001). Intuitively, the method can be described as follows. Every node of the search tree maintained by a solver (say, Forward Checking) is associated with a CSP. When the algorithm considers the current node  $A$  of the search tree, it checks whether there is a dead-end node  $B$ , such that the CSP associated with  $A$  can be transformed by some symmetry to a CSP associated with  $B$  (with possible restriction of domains). If such a node  $B$  is found, the current node  $A$  is rejected without further exploration.

SBDD can be naturally transformed into a pruning method that works for general CSPs and does not require any prior knowledge about symmetries of the problem at hand. Instead of associating a node of the search tree with a CSP, it can be associated with filtered domains of unassigned variables. Intuitively, the current node of the search tree can be discarded if the current domains of the unassigned variables are subsets of the domains of the corresponding variables associated with some dead-end node. However, this method has little pruning effect for general CSPs. The reason is that it is very unlikely that the inclusion relation is satisfied for all the unassigned variables. In addition, the checking of the inclusion relation requires overhead which affects the runtime of the algorithm.

We observe that in order to reject the current node  $A$  of a search tree because of its "similarity" to a dead-end node  $B$ , there is no need to check all the unassigned variables. Instead, one can identify for every dead-end node a subset of unassigned variables that "certify" its failing. This subset is frequently quite small. We call it a *kernel*. Now, to discard the current node  $A$ , it is sufficient to verify the inclusion condition only for the kernel associated with a node  $B$ .

The present paper proposes a pruning method based on the idea. We call the pruning method RKP, which is an abbreviation of Responsibility sets, Kernels, Propagation (responsibility set is an intermediate structure used for computing of kernels). We combine the pruning technique with Forward Checking (FC) (Prosser, 1993) and Maintaining Arc-consistency (MAC) (Sabin and Freuder, 1994) algorithms. Accordingly, we call the obtained algorithms FC-RKP and MAC-RKP, respectively.

We provide both theoretical and empirical evaluation of the proposed method. In the theoretical part, we prove that under certain restrictions, FC-RKP exactly simulates FC-CBJ (Prosser, 1993) and show that without these restrictions, the RKP-technique naturally generalizes pruning by the use of conflict sets. This result is interesting

because the information kept by the structures is very different: responsibility sets and kernels register information about "future conflicts" (with unassigned variables), while conflict sets register information about past conflicts (nogoods). A nice consequence of the result is that it builds a bridge between such apparently different areas of constraint reasoning as methods of symmetry breaking and "intelligent backtracking".

The empirical evaluation shows that the proposed technique has better pruning abilities than pruning by the use of conflict sets. In particular, we compare MAC-RKP and MAC-CBJ on two benchmark problems: random CSPs and graph coloring problems. According to our experiments, MAC-RKP is faster than MAC-CBJ on problems with low density by a factor of 2 to 4.5. As instances become denser, MAC-RKP saves less computational effort, but even for very dense CSPs MAC-RKP produces smaller search trees than MAC-CBJ. Moreover, being carefully implemented with the use of memorization techniques, MAC-RKP takes less runtime than MAC-CBJ over the whole range of densities. For graph coloring problems the results are even better. MAC-RKP produces much smaller search trees than MAC-CBJ for the whole range of graph densities. Accordingly, it takes much less runtime.

The rest of the paper is organized as follows. Section 2 presents the terminology. Section 3 and 4 describe FC-RKP and MAC-RKP, respectively. Section 5 presents the experimental evaluation. Section 6 concludes the paper by discussing possible directions of further research.

## 2. Terminology

The present paper considers binary CSP. The model of a binary CSP is a binary constraint network (CN). A CN  $Z$  consists of three parts. The first part is a set of variables. Every variable has a domain of values. We denote a value  $val$  of a variable  $v$  by  $\langle v, val \rangle$ . The set of domains of variables is the second part of  $Z$ . A pair of values of different variable is either *compatible* or incompatible (*conflicting*). The set of all compatible pairs of values of a pair of variables  $u$  and  $v$  is called the *constraint* between  $u$  and  $v$ . The set of all constraints is the third part of  $Z$ .

A set  $P$  of values of different variables is *consistent* (*satisfies* all the constraints) if all the values of  $P$  are mutually compatible. In this case, we call  $P$  a *partial solution* of  $Z$ . Let  $\langle u, val \rangle \in P$ . Then we say that  $P$  *assigns*  $u$ . Accordingly,  $\langle u, val \rangle$  is the *assignment* of  $u$  in  $P$ . If  $P$

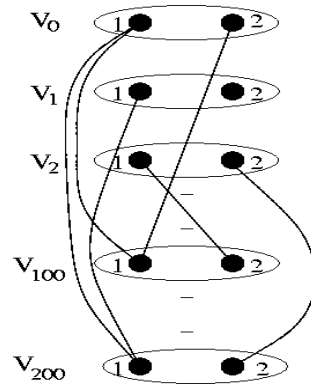


Figure 1. A constraint network.

assigns all the variables, it is a *solution* of  $P$ . The task of CSP is to find a solution of  $Z$  or to derive that no solution exists.

Generally, not every partial solution is a subset of a full solution. If  $P$  is not a subset of any solution, it is called a *nogood*. Note that sometimes in the literature, the notion of nogood has a broader meaning: it includes also set of assignments with inner conflicts. We emphasize that in the present paper a nogood is always consistent.

Finally, we extend the notion of compatibility. A value  $\langle u, val \rangle$  is *compatible* with a partial solution  $P$  if the following two conditions hold:

- if  $u$  is assigned by  $P$  then  $\langle u, val \rangle \in P$ ;
- $\langle u, val \rangle$  is compatible with all the assignments of  $P$ .

### 3. From FC to FC-RKP

#### 3.1. DESIGN OF THE ALGORITHM

In this section we describe the FC-RKP algorithm. We present it by step-by-step transformation of Forward Checking (FC) (Prosser, 1993). The CN in Figure 1 serves us as the working example.

We start the description of the proposed algorithm from definition of a responsibility set.

**DEFINITION 1.** *Let  $P$  be a nogood of a CN  $Z$ . A set  $S$  of variables is a responsibility set of  $P$  if there is no consistent extension of  $P$  that assigns all the variables of  $S$ .*

For the CN in Figure 1, let  $P = \{\langle v_0, 1 \rangle\}$ . A responsibility set of  $P$  is  $\{v_2, v_{100}, v_{200}\}$ .

Next, we modify FC so that it computes a responsibility set for every discovered nogood.

Imagine that at some moment of its work, FC recognizes that the current partial solution  $P$  is a nogood. Let  $\langle u, val \rangle \in P$  be the last (chronologically) assignment of  $P$ . FC removes  $\langle u, val \rangle$  from  $P$  and from the current domain of  $u$ . We say that  $P$  is the nogood associated with  $\langle u, val \rangle$  and denote it by  $nogood(\langle u, val \rangle)$ .

The modified version of FC associates every removed value  $\langle u, val \rangle$  with a set denoted by  $rs(\langle u, val \rangle)$ . To do this, every time when a value  $\langle u, val \rangle$  is removed, the modified FC runs a procedure  $compute\_rs(\langle u, val \rangle)$  which computes  $rs(\langle u, val \rangle)$ . The procedure is described in Algorithm 1. Further we prove that whenever  $\langle u, val \rangle$  is removed by backtrack,  $rs(\langle u, val \rangle)$  is a responsibility set of  $nogood(\langle u, val \rangle)$ . We term the resulting version of Forward Checking FC-R.

---

**Algorithm 1**  $compute\_rs(\langle u, val \rangle)$

---

- 1: **if**  $\langle u, val \rangle$  is removed by backtrack **then**
  - 2:   Let  $v$  be an unassigned variable whose empty current domain caused the backtrack
  - 3:   Let  $S$  be the union of  $rs$ -sets of all values of  $v$
  - 4:    $rs(\langle u, val \rangle) \rightarrow S \cup \{v\}$
  - 5: **else**
  - 6:    $rs(\langle u, val \rangle) \rightarrow \emptyset$
  - 7: **end if**
- 

To illustrate the work of the  $compute\_rs$  procedure, we simulate a few steps of the run of FC-R on the CN in Figure 1. Assume that after the first three assignments, the current partial solution is  $\{\langle v_0, 1 \rangle, \langle v_1, 1 \rangle, \langle v_2, 1 \rangle\}$ . After the first assignment,  $\langle v_{100}, 1 \rangle$  is removed and associated with the empty  $rs$ -set (line 6 of Algorithm 1). After the last assignment,  $\langle v_{100}, 2 \rangle$  is removed and also associated with the empty  $rs$ -set. Thus, after the first three assignments, the current domain of  $v_{100}$  is emptied and FC-R has to backtrack. It removes  $\langle v_2, 1 \rangle$  and sets  $rs(\langle v_2, 1 \rangle)$  to  $\{v_{100}\}$  (lines 2-4 of Algorithm 1). The next current partial solution is  $\{\langle v_0, 1 \rangle, \langle v_1, 1 \rangle, \langle v_2, 2 \rangle\}$ . This solution empties the current domain of  $v_{200}$ , so  $\langle v_2, 2 \rangle$  is removed and  $rs(\langle v_2, 2 \rangle)$  is set to  $\{v_{200}\}$ . Next, the current domain of  $v_2$  is emptied. The union of the  $rs$ -sets of the removed values of  $v_2$  is  $\{v_{100}, v_{200}\}$ . By lines 3-4 of Algorithm 1,  $rs(\langle v_1, 1 \rangle)$  is set to  $\{v_2, v_{100}, v_{200}\}$ . It is worth noting that Algorithm 1 is a slight

modification of the method described in (Schiex and Verfaillie, 1993): instead of a set of "culprit" constraints which can take quadratic space, FC-R finds a set of "culprit" future variables.

Next, we introduce the definition of kernel.

**DEFINITION 2.** *Let  $P$  be a nogood of a CN  $Z$  and let  $S$  be a responsibility set of  $P$ . Let  $\langle u, val \rangle \in P$ . The kernel of  $\langle u, val \rangle$  with respect to  $P$  and  $S$  is a set  $K \subseteq S$  that contains all the variables of  $S$  having in their domains values that are incompatible with  $\langle u, val \rangle$ , but compatible with the other values of  $P$ .*

To illustrate the notion of a kernel, consider the partial solution  $P = \{\langle v_0, 2 \rangle, \langle v_1, 1 \rangle\}$ . It is a nogood with a responsibility set  $S = \{v_2, v_{100}, v_{200}\}$ , but the kernel of  $\{\langle v_1, 1 \rangle\}$  with respect  $P$  and  $S$  is  $\{v_{200}\}$ .

We further modify FC-R so that every time when a value  $\langle u, val \rangle$  is deleted by backtrack, it is associated with a set  $ker(\langle u, val \rangle)$  which is the kernel of  $\langle u, val \rangle$  with respect to  $nogood(\langle u, val \rangle)$  and  $rs(\langle u, val \rangle)$ . In particular, we add to FC-R the following two operations:

- When a value  $\langle u, val \rangle$  is added to the current partial solution, keep in memory the set of unassigned variables that have in their current domains values incompatible with  $\langle u, val \rangle$  (it is easy to do, simply remember the values of which variables have been removed in the respective check-forward stage);
- When  $\langle u, val \rangle$  is removed from the current partial solution, set  $ker(\langle u, val \rangle)$  to be the intersection of the set described in the previous item with  $rs(\langle u, val \rangle)$ .

We call the resulting algorithm FC-RK. Note that the two-step computation of  $ker$ -set avoids recomputation of values incompatible with  $\langle u, val \rangle$ . These values are computed anyway when  $\langle u, val \rangle$  is added to the current partial solution, so it is easy to keep in memory the variables to whose domains the values belong and then simply to compute the intersection.

The following lemma is central for the proposed pruning method.

**LEMMA 1.** *Let  $P$  be a nogood that contains a value  $\langle u, val \rangle$ . Let  $S$  and  $K$  be a responsibility set of  $P$  and the kernel of  $\langle u, val \rangle$  with respect to  $P$  and  $S$ , respectively. Let  $T$  be a partial solution compatible with  $P$  that assigns all the variables of  $K$ . Then  $(P \setminus \{\langle u, val \rangle\}) \cup T$  is a nogood with a responsibility set  $S \setminus K$ .*

**Proof.** Assume by contradiction that the statement of the lemma is not true. Then there is a partial solution  $P'$  assigning all the variables of  $S$ , containing  $T$ , and compatible with  $P \setminus \{\langle u, val \rangle\}$ . Note that  $P'$  is compatible also with  $\langle u, val \rangle$ . Really,  $T$  is compatible with  $\langle u, val \rangle$  by definition and  $P' \setminus T$  is compatible with  $\langle u, val \rangle$  because it assigns variables that are not in the kernel of  $\langle u, val \rangle$  with respect to  $P$  and  $S$ . Thus we get that  $P'$  is compatible with  $P$ , a contradiction with Definition 1. ■

To explain the proposed pruning method, we state Lemma 1 again in a more informal setting. Suppose we have a nogood  $P$  with  $\langle u, val \rangle \in P$ . It might be that  $P \setminus \{\langle u, val \rangle\}$  is not a nogood that is it can be extended to a solution of the underlying CN. Lemma 1 states a necessary condition for possibility of such extension. According to this condition, every solution that contains *assigns at least one variable of  $\ker(\langle u, val \rangle)$  with a value incompatible with  $\langle u, val \rangle$* . The precise statement of Lemma 1 is even stronger: it states that the condition above must hold for every partial solution that contains  $P \setminus \{\langle u, val \rangle\}$  and assigns all the variables of the responsibility set of  $P$ . Speaking in terms of constraints, Lemma 1 imposes a constraint on the variables of the kernel of  $\langle u, val \rangle$ . This constraint can be used by FC-RK to perform additional pruning.

To illustrate Lemma 1 on our working example, let  $P = \{\langle v_0, 1 \rangle\}$ , let  $\langle u, val \rangle = \langle v_0, 1 \rangle$ , let  $S = \{v_2, v_{100}, v_{200}\}$  (then  $K = \{v_{100}, v_{200}\}$ ), and let  $T = \{\langle v_{100}, 2 \rangle, \langle v_{200}, 2 \rangle\}$ . Obviously,  $P \setminus \{\langle u, val \rangle\} \cup T$  (which equals  $T$  in our case) is a nogood with a responsibility set  $v_2 = S \setminus K$ .

Lemma 1 has the following two useful corollaries:

**COROLLARY 1.** *If in Lemma 1  $|K| = 0$  then  $P \setminus \{\langle u, val \rangle\}$  is a nogood with a responsibility set  $S$ .*

**COROLLARY 2.** *If in Lemma 1  $|K| = 1$  and  $v$  is the only variable contained in  $K$  then for any  $\langle v, val' \rangle$  compatible with  $P$ ,  $P \setminus \{\langle u, val \rangle\} \cup \{\langle v, val' \rangle\}$  is a nogood with a responsibility set  $S \setminus \{v\}$ .*

To illustrate Corollary 1, let  $P = \{\langle v_0, 1 \rangle, \langle v_1, 1 \rangle\}$ . Note that  $S = \{v_2, v_{100}, v_{200}\}$  is a responsibility set of  $P$ . Let  $\langle u, val \rangle = \langle v_1, 1 \rangle$ . Obviously the kernel of  $\langle u, val \rangle$  with respect to  $P$  and  $S$  is  $\emptyset$ . Observe that indeed,  $P \setminus \{\langle u, val \rangle\} = \{\langle v_0, 1 \rangle\}$  is a nogood with a responsibility set  $S$ .

To illustrate Corollary 2, let  $P = \{\langle v_0, 2 \rangle, \langle v_1, 1 \rangle\}$ . Note that  $S = \{v_2, v_{100}, v_{200}\}$  is a responsibility set of  $P$ . Take  $\langle u, val \rangle$  to be  $\langle v_1, 1 \rangle$ . Observe that the kernel of  $\langle u, val \rangle$  with respect to  $P$  and  $S$  is  $v_{200}$ . Replace  $\langle v_1, 1 \rangle$  by  $\langle v_{200}, 2 \rangle$  in  $P$ . Observe that the resulting partial solution is a nogood with a responsibility set  $\{v_2, v_{100}\}$ .

We propose a modification of FC-RK which we term FC-RKP ('P' stands for "Propagation"). FC-RKP has two additional procedures called *propagate\_backward* and *propagate\_forward*.

REMARK 1. *These procedures perform additional filtering of values. Therefore, it is important to point out when the values removed by the procedures are restored in their current domains. We assume that FC-RKP has a usual for FC policy of restoring of removing values. That is, let  $P$  be the current partial solution at the time of removing a value  $\langle u, val \rangle$ . Then  $\langle u, val \rangle$  is restored in the current domain of  $u$  when  $P$  is discarded.*

The procedure *propagate\_backward* is applied to a value  $\langle u, val \rangle$  whenever it is removed from the current partial solution. The procedure runs next after *compute\_rs*( $\langle u, val \rangle$ ). Algorithm 2 describes the procedure.

---

**Algorithm 2** *propagate\_backward*( $\langle u, val \rangle$ )

---

```

1: if  $|ker(\langle u, val \rangle)| = 0$  then
2:   discard_assignment( $rs(\langle u, val \rangle)$ )
3: end if
4: if  $|ker(\langle u, val \rangle)| = 1$  then
5:   Let  $v$  be the variable contained in  $ker(\langle u, val \rangle)$ 
6:   for every value  $val'$  of the current domain of  $v$  that is compatible
       with  $\langle u, val \rangle$  do
7:     Set nogood( $\langle v, val' \rangle$ ) to the union of the current partial
       solution and  $\{\langle v, val' \rangle\}$ 
8:      $rs(\langle v, val' \rangle) \leftarrow rs(\langle u, val \rangle) \setminus \{v\}$ 
9:     Remove  $\langle v, val' \rangle$  from the current domain of  $v$ 
10:  end for
11: end if

```

---

Algorithm 2 performs additional pruning operations if the size of  $ker(\langle u, val \rangle)$  is either 0 or 1. In the former case, the current partial solution is a nogood according to Corollary 1 therefore the procedure performs backtrack. However, the backtrack operations relevant to the case are slightly different from usual backtrack. In particular, the *rs*-set of the assignment being discarded is not computed by procedure *compute\_rs*. Therefore, we place these operations into a separate procedure and term it *discard\_assignment*. Algorithm 3 describes the procedure.

First of all, the procedure *discard\_assignment* checks whether the current partial solution is empty. If yes, the solver stops and returns "NO SOLUTION". Otherwise, the procedure discards the last assignment of the current partial solution and performs a number of additional operations that are necessary in order to preserve consistency. In particular,  $nogood(\langle v, val' \rangle)$  is set to be equal to the current partial solution,  $rs(\langle v, val' \rangle)$  is assigned with a set the procedure gets as a parameter. In the case the procedure is called from *propagate\_backward*, the set is  $rs(\langle u, val \rangle)$ . Further,  $ker(\langle v, val' \rangle)$  is computed as described for FC-RK, the values that were removed when  $\langle v, val' \rangle$  had been added to the the current partial solution are restored in their current domains, the value  $\langle v, val' \rangle$  is removed from the current partial solution and from the current domain of  $v$ . Finally, the procedure *propagate\_backward* is called recursively, because there may be a sequence of removed values with the *ker*-sets of size 0.

Let us return to the code of *propagate\_backward*. If  $ker(\langle u, val \rangle)$  has size 1, the procedure removes from the current domain of  $v$  (the only variable that belongs to  $ker(\langle u, val \rangle)$ ) all values that are compatible with  $\langle u, val \rangle$ . Such removing is possible according to Corollary 2. The removed values are associated with the appropriate *nogood*-sets and *rs*-sets. Note that we do not compute the *ker*-sets for the removed values, because *ker*-sets are computed only for discarded assignments of the current partial solution.

---

**Algorithm 3** *discard\_assignment*( $RS$ )

---

```

1: if The current partial solution is empty then
2:   Stop FC-RKP and return "NO SOLUTION"
3: else
4:   Let  $\langle v, val' \rangle$  be the last assignment of the current partial solution
5:   Set  $nogood(\langle v, val' \rangle)$  to the current partial solution
6:    $rs(\langle v, val' \rangle) \leftarrow RS$ 
7:   Compute  $ker(\langle v, val' \rangle)$ 
8:   Restore in the current domains of unassigned variables all values
   that were removed as a result of appending  $\langle v, val' \rangle$  to the current
   partial solution
9:   Remove  $\langle v, val' \rangle$  from the current partial solution and from the
   current domain of  $v$ 
10:  propagate_backward( $\langle v, val' \rangle$ )
11: end if

```

---

The procedure *propagate\_forward* is applied whenever a new assignment is appended to the current partial solution. It runs just after the completion of the lookahead procedure. Algorithm 4 describes the procedure. The following notion is used in the description of the procedure.

**DEFINITION 3.** *A variable  $v$  is neutral with respect to a value  $\langle u, val \rangle$  if at least one of the following two conditions holds:*

- $v$  is assigned a value that is compatible with  $\langle u, val \rangle$ ;
- all values of the current domain of  $v$  are compatible with  $\langle u, val \rangle$ .

The main loop of procedure *propagate\_forward* (lines 2-18) scans all values that have kernels associated with them. If the procedure checks a value  $\langle u, val \rangle$  and observes that all variables of  $ker(\langle u, val \rangle)$  are neutral with respect to  $\langle u, val \rangle$ , the procedure initiates a backtrack (lines 7 and 8). If the *ker*-set of  $\langle u, val \rangle$  has exactly one variable  $v$  which is not neutral with  $\langle u, val \rangle$ , the procedure deletes from the domain of  $v$  all values that are compatible with  $\langle u, val \rangle$  (lines 10-14).

Note that the procedure has the external while-loop that can run the inner cycle a number of times. The procedure exits from the loop when the last iteration has not removed any value. To demonstrate that the scanning loop can run more than one time consider the following scenario.

- The *ker*-set of a removed value  $\langle u_1, val_1 \rangle$  has exactly one variable  $v$  which is not neutral with respect to  $\langle u_1, val_1 \rangle$
- The *ker*-set of a removed value  $\langle u_2, val_2 \rangle$  has exactly two variables  $v$  and  $w$  which are not neutral with respect to  $\langle u_2, val_2 \rangle$
- After the removing of the values compatible with  $\langle u_1, val_1 \rangle$  from the current domain of  $v$ , variable  $v$  becomes neutral with respect to  $\langle u_2, val_2 \rangle$ .
- $\langle u_2, val_2 \rangle$  is scanned before  $\langle u_1, val_1 \rangle$ .

According to the scenario, in the first iteration the algorithm prunes the current domain of  $v$  and in the second iteration it prunes the current domain of  $w$ .

We demonstrate on the working example a case where the *propagate\_forward* procedure can perform pruning. Assume that the current partial solution is  $\{\langle v_0, 2 \rangle\}$ , that  $\langle v_0, 1 \rangle$  is removed, and that  $ker(\langle v_0, 1 \rangle) = \{v_{100}, v_{200}\}$ .

---

**Algorithm 4** *propagate\_forward()*


---

```

1: while true do
2:   for every removed value  $\langle u, val \rangle$  that is associated with a ker-set
   do
3:     if  $ker(\langle u, val \rangle)$  has at most one variable that is not neutral
       with  $\langle u, val \rangle$  then
4:       Let NS be the set of all values incompatible with  $\langle u, val \rangle$ 
       that belong to the domains of variables having the following
       properties:
       – unassigned;
       – belong to  $ker(\langle u, val \rangle)$ ;
       – neutral with respect to  $\langle u, val \rangle$ .

5:       Let RSET be the union of rs-sets of all values of NS
6:       if all variables of  $kernel(\langle u, val \rangle)$  are neutral with respect
       to  $\langle u, val \rangle$  then
7:         Apply discard_assignment( $unrs \cup RSET$ ), where unrs is
       the unassigned subset of  $rs(\langle u, val \rangle)$ 
8:         Return
9:       else
10:        Let v be the only variable of  $ker(\langle u, val \rangle)$  that is not
        neutral with respect to  $\langle u, val \rangle$ 
11:        for every value val' of the current domain of v that is
        compatible with  $\langle u, val \rangle$  do
12:          Set nogood( $\langle v, val' \rangle$ ) to the union of the current partial
          solution and  $\{\langle v, val' \rangle\}$ 
13:           $rs(\langle v, val' \rangle) \leftarrow unrs \setminus \{v\} \cup RSET$ , where unrs is the
          unassigned subset of  $rs(\langle u, val \rangle)$ 
14:          Remove  $\langle v, val' \rangle$  from the current domain of v
15:        end for
16:      end if
17:    end if
18:  end for
19:  if No values have been removed in the iteration then
20:    Return
21:  end if
22: end while

```

---

Observe that  $v_{100}$  is neutral with respect to  $\langle v_0, 2 \rangle$ . The only non-neutral variable is  $v_{200}$ . The procedure executes lines 10-15 and removes  $\langle v_{200}, 2 \rangle$  from the current domain of  $v_{200}$ .

### 3.2. CORRECTNESS PROOF

The correctness of the FC-RKP algorithm is justified by the following two theorems.

**THEOREM 1.** *When FC-RKP removes a value  $\langle u, val \rangle$  from the current domain of  $u$ , at least one of the following conditions holds:*

- $\langle u, val \rangle$  is incompatible with the last assignment of the current partial solution and  $rs(\langle u, val \rangle) = \emptyset$ ;
- $\langle u, val \rangle$  is compatible with the current partial solution, and  $rs(\langle u, val \rangle)$  is a responsibility set of  $nogood(\langle u, val \rangle)$ .

**Proof.** For the case where  $\langle u, val \rangle$  is incompatible with the current partial solution the proof is easy: simply check the description of procedure *compute\_rs* (Algorithm 1).

For the other case, we consider a chronological sequence of events of removing of values compatible with the current partial solution at the time of their removal. The theorem is proved by induction on the length of the sequence.

Let  $\langle u, val \rangle$  be the value associated with the first item of the sequence. The only possible scenario of removal of the value is that  $\langle u, val \rangle$  is the last assignment of the current partial solution. The value is removed by backtrack and the backtrack is caused by an unassigned variable  $v$  all values of which are in conflict with assignments of the current partial solution.

Observe that in this case  $rs(\langle u, val \rangle) = \{v\}$  and it is indeed a responsibility set of  $nogood(\langle u, val \rangle)$ .

Now, consider an event which is not the first in the chronological sequence. Let  $\langle u, val \rangle$  be the value removed by the event and assume that the theorem holds for all values associated with the previous events. The value  $\langle u, val \rangle$  can be removed by one of the following five ways:

1. by backtrack;
2. by procedure *discard\_assignment* called from line 2 of *propagate\_backward*;
3. in lines 7-9 of *propagate\_backward*;
4. by procedure *discard\_assignment* called from line 7 of *propagate\_forward*;

5. in lines 12-14 of *propagate\_forward*.

Consider these possibilities one by one. Assume  $\langle u, val \rangle$  is removed by backtrack. Then the backtrack is caused by the empty current domain of some unassigned variable  $v$ . The procedure *compute\_rs* sets  $rs(\langle u, val \rangle)$  to  $S \cup \{v\}$ , where  $S$  is the union of  $rs$ -sets of all values of  $v$ . If all values of  $v$  are incompatible with the current partial solution we have the same situation as with the first item in the sequence. Otherwise, let  $\langle v, val' \rangle$  be a value compatible with the current partial solution and let  $P$  be a partial solution obtained by appending  $\langle v, val' \rangle$  to the current partial solution. Observe that  $nogood(\langle v, val' \rangle) \subseteq P$ . Note that by the induction assumption,  $rs(\langle v, val' \rangle)$  is a responsibility set of  $nogood(\langle v, val' \rangle)$  therefore  $rs(\langle v, val' \rangle)$  is a responsibility set of  $P$ . Taking into account that  $rs(\langle v, val' \rangle) \subseteq S$ , we get that  $P$  cannot be extended to a partial solution that assigns all the values of  $S$ .

It follows that the current partial solution (that equals  $nogood(\langle u, val \rangle)$ ) does not have a consistent extension that assigns all the variables of  $S \cup \{v\}$ . In other words  $rs(\langle u, val \rangle)$  is a responsibility set of  $nogood(\langle u, val \rangle)$ .

For the second case, when  $\langle u, val \rangle$  is deleted by procedure *discard\_assignment* called from line 5 of *propagate\_backward*, the theorem follows from Corollary 1.

For the third case when  $\langle u, val \rangle$  is deleted in lines 10-12 of *propagate\_backward*, the theorem follows from Corollary 2.

Consider the fourth case, where  $\langle u, val \rangle$  is deleted by the procedure *discard\_assignment* called from line 7 of *propagate\_forward*. In this case, there is some removed value  $\langle v, val' \rangle$  such that every variable of its  $ker$ -set is neutral with respect to it. Denote the current partial solution by  $P$ . Let  $NS$  be the set defined in line 4 of procedure *propagate\_forward*. That is,  $NS$  is the set of all values incompatible with  $\langle v, val' \rangle$  that belong to the domains of variables with the following properties:

- unassigned;
- belong to  $ker(\langle v, val' \rangle)$ ;
- neutral with respect to  $\langle v, val' \rangle$ .

Let  $RSET$  be the union of  $rs$ -sets of all values of  $NS$ .

Let  $T$  be a partial solution compatible with  $P$  that assigns all the unassigned variables of  $ker(\langle v, val' \rangle)$  (if such a partial solution does not exist, the theorem follows immediately). Assume first that there is a value  $\langle w, dval \rangle \in T \cap NS$ . In this case,  $rs(\langle w, dval \rangle)$  is a responsibility set of  $P \cup T$ . Really, on the one hand,  $rs(\langle w, dval \rangle)$  is a responsibility set of  $nogood(\langle w, dval \rangle)$  by the induction assumption; on the other hand,

$nogood(\langle w, dval \rangle) \subseteq P \cup T$ . Taking into account that  $rs(\langle w, dval \rangle) \subseteq RSET$ , we derive that there is no consistent extension of  $P \cup T$  that assigns all the variables of  $RSET$ .

For the case when  $T$  does not intersect with  $NS$ , let  $T'$  be the superset of  $T$  compatible with  $P$  that assigns all the variables of  $ker(\langle v, val' \rangle)$ . (By definition,  $T$  assigns only those variables of  $ker(\langle v, val' \rangle)$  that are not assigned by  $P$ , hence  $T'$  is the union of  $T$  and those assignments of variables of  $ker(\langle v, val' \rangle)$  that are contained in  $P$ .)

Observe that  $T'$  is compatible with  $nogood(\langle v, val' \rangle)$ . Really,  $T'$  is compatible with  $\langle v, val' \rangle$  which follows from the "neutrality" of all variables of  $ker(\langle v, val' \rangle)$  with respect to  $\langle v, val' \rangle$ . Also, by definition,  $T'$  is compatible with  $P \setminus T'$  which is a superset of  $nogood(\langle v, val' \rangle) \setminus \{\langle v, val' \rangle\}$ .

Taking into account that  $rs(\langle v, val' \rangle)$  is a responsibility set of  $nogood(\langle v, val' \rangle)$  and applying Lemma 1, we get that  $nogood(\langle v, val' \rangle) \cup T'$  is a nogood with a responsibility set  $rs(\langle v, val' \rangle) \setminus ker(\langle v, val' \rangle)$ . Of course, the same is true regarding  $P \cup T$ , which is a superset of  $nogood(\langle v, val' \rangle) \cup T'$ .

To summarize, we derived that  $P$  cannot be extended to a partial solution that assigns all the variables of  $unrs \cup RSET$ , where  $unrs$  is the unassigned subset of  $rs(\langle v, val' \rangle)$ . Thus  $P = nogood(\langle u, val \rangle)$  is a nogood with a responsibility set  $unrs \cup RSET$  which proves the fourth case of the theorem.

For the last case where  $\langle u, val \rangle$  is deleted in lines 12-14 of procedure *propagate\_forward*, there is a removed value  $\langle v, val' \rangle$  whose *ker*-set has exactly one variable  $w$  which is not neutral with respect to  $\langle v, val' \rangle$ . Denote the current partial solution by  $P$ . Let  $\langle w, dval \rangle$  be a value compatible with  $\langle v, val' \rangle$ . Appending  $P$  to the current partial solution, we get the fourth case of the theorem. Accordingly,  $nogood(\langle u, val \rangle) = P \cup \{\langle u, val \rangle\}$  is a nogood with a responsibility set  $unrs \cup RSET$ , which proves the validity of the last case of the theorem. ■

**THEOREM 2.** *FC-RKP is sound, complete and always terminates in a finite number of steps.*

**Proof.** FC-RKP can be seen as FC with additional filtering, therefore soundness and termination of FC-RKP follow from soundness and termination of FC (Kondrak and van Beek, 1995). The completeness follows from Theorem 1. ■

### 3.3. COMPLEXITY ANALYSIS

Let us discuss an implementation of procedure *propagate\_forward* that reduces its worst-case complexity. First of all, observe why the straightforward implementation of Algorithm 4 is inapplicable. Denote

the number of variables of the underlying CN and the maximal domain size by  $n$  and  $m$ , respectively. Then there are at most  $n*m$  removed values associated with kernels. The kernel of every removed value has size  $O(n)$ . Checking neutrality of a variable takes  $O(m)$ . In addition, note that checking neutrality for the same variable can repeat  $O(n*m)$  times because of the external loop. Summarizing all the above measures, we get a complexity of  $O(n^3 * m^3)$  which is too much for a consistency maintenance procedure.

Let  $\langle u, val \rangle$  be a removed value associated with a kernel. Observe that neutrality checking for an assigned variable of  $ker(\langle u, val \rangle)$  is different from the checking for an unassigned variable. In the former case, it is just one compatibility check. In the latter case,  $O(m)$  compatibility checks must be performed. Let us call the neutrality condition for an assigned variable the *feasibility condition* and for an unassigned variable the *inclusion condition*. Accordingly,  $\langle u, val \rangle$  is *feasible* if the feasibility condition is satisfied for all the assigned variables of  $ker(\langle u, val \rangle)$ .

Note that if  $\langle u, val \rangle$  is infeasible, it cannot become feasible during the iterations of the external loop of Algorithm 4, because the loop cannot change the assignment that violates the feasibility condition. Therefore, in our implementation, the feasibility of the removed values is checked *before* the main loop and in the main loop the inclusion condition is checked only for values that are *known to be feasible*. This modification does not reduce the worst-case complexity, but essentially reduces the computational effort spent for feasibility checking. In particular, the complexity of feasibility checking is reduced from  $O(n^3m^2)$  to  $O(n^2m)$ .

To reduce the complexity of checking the inclusion condition, observe that if the inclusion condition is satisfied for a variable  $v \in ker(\langle u, val \rangle)$  at some node of the search tree, it will be satisfied for all descendants of the node because current domains of variables cannot grow as the length of the current partial solution increases. More precisely, if at some node of the search tree a *subset* of values of  $v$  that violate the inclusion condition is eliminated from the current domain of  $v$ , the subset remains eliminated for all the descendants of that node. This observation suggests that we can *amortize* the inclusion condition checking among iterations of the main loop of *propagate\_forward* by keeping in memory the "place" where the inclusion condition has failed in the previous iteration and proceeding to check the condition directly from that place (if the condition has been satisfied then the current node of the search tree is pruned and no memorization is needed). Thus the complexity of the main loop equals the complexity of one iteration that is  $O(n^2m^2)$ . Taking into account the complexity of the feasibility checking, we get that  $O(n^2m^2)$  is the worst-case time complexity of the whole procedure.

#### 4. From MAC to MAC-RKP

In this section we show how to transform the Maintaining Arc-Consistency Algorithm (MAC) into MAC-RKP. We assume that the procedure for achieving arc-consistency uses the notion of *support* (Mohr and Henderson, 1986). The support of a value  $\langle u, val \rangle$  with respect to a variable  $v$  is the number of values of  $v$  compatible with  $\langle u, val \rangle$ . If the support is 0,  $\langle u, val \rangle$  is removed.

The process of transformation of MAC into MAC-RKP is very similar to the transformation of FC to FC-RKP described in the previous section. We start by presenting the algorithm MAC-R. Recall that this algorithm runs a procedure *compute\_rs* every time when a value  $\langle u, val \rangle$  is deleted from the current domain of  $u$ . The procedure *compute\_rs* for MAC-R is slightly different from the procedure for FC-R, because it takes into account additional possibility of value removal during the imposing arc-consistency. Algorithm 5 describes the procedure. To avoid confusion, we call the procedure *compute\_rs\_mac*.

---

**Algorithm 5** *compute\_rs\_mac*( $\langle u, val \rangle$ )

---

```

1: if  $\langle u, val \rangle$  is removed by backtrack then
2:   Let  $v$  be an unassigned variable whose empty current domain
   caused the backtrack
3:   Let  $S$  be the union of rs-sets of all values of  $v$ 
4:    $rs(\langle u, val \rangle) \rightarrow S \cup \{v\}$ 
5: else
6:   if  $\langle u, val \rangle$  is removed during arc-consistency achieving then
7:     Let  $v$  be the variable with respect to which  $\langle u, val \rangle$  has the
     zero support
8:     Let  $S$  be the union of rs-sets of all removed values of  $v$ 
9:      $rs(\langle u, val \rangle) \rightarrow S \cup \{v\}$ 
10:  else
11:     $rs(\langle u, val \rangle) \rightarrow \emptyset$ 
12:  end if
13: end if

```

---

The next step of the presented transformation is to formulate the MAC-RK algorithm. Analogously to FC-RK, MAC-RK computes *ker*-sets for values removed by backtrack. The computation of *ker*-sets is analogous to that of FC-RK. That is, when a value  $\langle u, val \rangle$  is appended to the current partial solution, the algorithm keeps in memory the set  $K$  of unassigned variables from which values have been removed as a result

of the assignment. When  $\langle u, val \rangle$  is deleted from the current partial solution,  $ker(\langle u, val \rangle)$  is set to the intersection of  $K$  with  $rs(\langle u, val \rangle)$ .

Now we prove an additional corollary of Lemma 1 which will be useful for the construction of MAC-RKP.

**COROLLARY 3.** *In Lemma 1, assume that  $|K| = 2$ . Let  $v$  and  $w$  be the two variables contained in  $K$ . Let  $\langle v, val' \rangle$  be a value of  $v$  compatible with  $P$  such that there is no value of  $w$  compatible with  $\langle v, val' \rangle$  but incompatible with  $P$ . Then  $P \setminus \{\langle u, val \rangle\} \cup \{\langle v, val' \rangle\}$  is a nogood with a responsibility set  $S \setminus \{v\}$ .*

**Proof.** Denote  $P \setminus \{\langle u, val \rangle\}$  by  $P'$ . Observe that for any  $\langle w, val'' \rangle$ , either  $\langle w, val'' \rangle$  is incompatible with  $\langle v, val' \rangle$  or  $P' \cup \{\langle v, val' \rangle, \langle w, val'' \rangle\}$  is a nogood with a responsibility set  $S \setminus \{u, v\}$ , by Lemma 1. In any case,  $P' \cup \{\langle v, val' \rangle\}$  cannot be extended to a partial solution that assigns all the variables of  $S \setminus \{v\}$ . Therefore, it is a nogood with a responsibility set  $S \setminus \{v\}$ . ■

To illustrate Corollary 3, assume that the CN in Figure 1 has an additional conflict between values  $\langle v_{100}, 2 \rangle$  and  $\langle v_{200}, 1 \rangle$ . Let  $P = \{\langle v_1, 1 \rangle\}$ ,  $\langle u, val \rangle = \langle v_1, 1 \rangle$ ,  $S = \{v_2, v_{100}, v_{200}\}$ . Then  $K = \{v_{100}, v_{200}\}$ . Observe that value  $\langle v_{100}, 2 \rangle$  must be deleted according to Corollary 3.

Now, we formulate the MAC-RKP algorithm. In particular, we introduce two additional procedures: *propagate\_backward\_mac* and *propagate\_forward\_mac*. These procedures are analogous of the *propagate\_backward* and *propagate\_forward* procedures for FC-RKP. Algorithm 6 presents the procedure *propagate\_backward\_mac*.

---

**Algorithm 6** *propagate\_backward\_mac*( $\langle u, val \rangle$ )

---

```

1: propagate_backward( $\langle u, val \rangle$ )
2: if  $|ker(\langle u, val \rangle)| = 2$  then
3:   Let  $v$  and  $w$  be the variables contained in  $ker(\langle u, val \rangle)$ 
4:   while true do
5:     propagate_pair( $v, w, \langle u, val \rangle, rs(\langle u, val \rangle)$ )
6:     propagate_pair( $w, v, \langle u, val \rangle, rs(\langle u, val \rangle)$ )
7:     impose_ac()
8:     if no value has been removed in the iteration then
9:       Return
10:    end if
11:  end while
12: end if

```

---

The procedure is applied to every value  $\langle u, val \rangle$  removed by back-track. It runs immediately after *compute\_rs\_mac* completes its run.

Lines 2-12 describe additional operations performed by the procedure. They are executed if the kernel of the removed value has size 2.

In this case the procedure achieves arc-consistency (lines 4-11). The arc-consistency achieving process is presented in a more complicated form than simply applying *impose\_ac()*. This is because Corollary 3 imposes additional conflicts between variables  $v$  and  $w$  contained in  $\ker(\langle u, val \rangle)$ . Therefore constraint propagation for the pair  $\{v, w\}$  is performed separately (lines 5,6) by application of the procedure *propagate\_pair* described in Algorithm 7. The procedure performs pruning as stated in Lemma 3.

---

**Algorithm 7** *propagate\_pair*( $v, w, \langle u, val \rangle, RS$ )

---

```

1: for every value  $\langle v, val' \rangle$  compatible with  $\langle u, val \rangle$  do
2:   if every value  $\langle w, val'' \rangle$  compatible with  $\langle v, val' \rangle$  is also compat-
     ible with  $\langle u, val \rangle$  then
3:     Remove  $\langle v, val' \rangle$  from the current domain of  $v$ 
4:     Set  $nogood(\langle v, val' \rangle)$  to the union of the current partial
     solution and  $\{\langle v, val' \rangle\}$ 
5:     Let  $S$  be the union of  $rs$ -sets of all removed values of  $w$ 
6:      $rs(\langle v, val' \rangle) \leftarrow S \cup \{v\} \cup RS$ 
7:   end if
8: end for

```

---

The procedure *propagate\_forward\_mac* is described in Algorithm 8. It is almost identical to the procedure *propagate\_forward* with the following exceptions: propagation in the case of kernel of size 2 (lines 17-20) and imposing of arc-consistency (line 23). The procedure is applied after every new assignment, just after finishing the lookahead procedure (imposing arc-consistency in our case).

Now we prove correctness of MAC-RKP. The proof is very similar to the proof of correctness of FC-RKP. For the sake of conciseness, in the case of analogy we refer the reader to the appropriate part of the correctness proof of FC-RKP.

**THEOREM 3.** *When MAC-RKP removes a value  $\langle u, val \rangle$  from the current domain of  $u$ , at least one of the following conditions holds:*

- $\langle u, val \rangle$  is incompatible with the last assignment of the current partial solution and  $rs(\langle u, val \rangle) = \emptyset$ ;
- $\langle u, val \rangle$  is compatible with the current partial solution, and  $rs(\langle u, val \rangle)$  is a responsibility set of  $nogood(\langle u, val \rangle)$ .

---

**Algorithm 8** *propagate\_forward\_mac()*


---

```

1: while true do
2:   for every removed value  $\langle u, val \rangle$  that is associated with a ker-set
   do
3:     if  $ker(\langle u, val \rangle)$  has at most two variables that are not neutral
       with  $\langle u, val \rangle$  then
4:       Let NS and RSET be the sets as defined in lines 4 and 5
       of Algorithm 4
5:       if all variables of  $kernel(\langle u, val \rangle)$  are neutral with respect
       to  $\langle u, val \rangle$  then
6:         Apply discard_assignment( $unrs \cup RSET$ ), where unrs is
         the unassigned subset of  $rs(\langle u, val \rangle)$ 
7:         Return
8:       end if
9:       if  $ker(\langle u, val \rangle)$  contains exactly one variable v that is not
       neutral with respect to  $\langle u, val \rangle$  then
10:        for Every value val' of the current domain of v that is
        compatible with  $\langle u, val \rangle$  do
11:          Set nogood( $\langle v, val' \rangle$ ) to the union of the current partial
          solution and  $\{\langle v, val' \rangle\}$ 
12:           $rs(\langle v, val' \rangle) \leftarrow unrs \setminus \{v\} \cup RSET$ , where unrs is the
          unassigned subset of  $rs(\langle u, val \rangle)$ 
13:          Remove  $\langle v, val' \rangle$  from the current domain of v
14:        end for
15:      end if
16:      if  $ker(\langle u, val \rangle)$  contains exactly two variables which are not
       neutral with respect to  $\langle u, val \rangle$  then
17:        propagate_pair(v, w,  $\langle u, val \rangle$ , RSET)
18:        propagate_pair(w, v,  $\langle u, val \rangle$ , RSET)
19:      end if
20:    end if
21:  end for
22:  impose_ac()
23:  if No value has been removed in the iteration then
24:    Return
25:  end if
26: end while

```

---

**Proof.** The first case, when  $\langle u, val \rangle$  is incompatible with the last assignment of the current partial solution, is trivial: the theorem follows from the description of procedure *compute\_rs\_mac*.

For the other case, like in the proof of Theorem 1, we consider a chronological sequence of events of removal of values that are compatible with the current partial solution and prove the theorem by induction on the length of the sequence.

Let  $\langle u, val \rangle$  be the value associated with the first item of the sequence. There are two possible ways of removal of the value. The first is removing by backtrack and it is analogous to the initial step of the induction considered in Theorem 1. The value can also be removed during arc-consistency achieving. In this case imagine that the assignment  $\langle u, val \rangle$  is added to the current partial solution. The current domain of the unsupported variable becomes empty and we get the backtrack case.

Assume now that  $\langle u, val \rangle$  is not the first item in the sequence and that the theorem holds for all the previous items. Similarly to Theorem 1, we consider all possibilities of removal of  $\langle u, val \rangle$ . The first five cases are identical to those of Theorem 1, as well as their proofs.

The additional cases are the following:

- the value  $\langle u, val \rangle$  is removed during achieving of arc-consistency;
- $\langle u, val \rangle$  is removed by the *propagate\_pair* procedure applied by *propagate\_backward\_mac*;
- $\langle u, val \rangle$  is removed by the *propagate\_pair* procedure applied by *propagate\_forward\_mac*.

For the first case, assume that  $\langle u, val \rangle$ , is appended to the current partial solution instead of being removed. Then the current domain for the unsupported variable becomes empty and we get the backtrack case of the induction step of Theorem 1.

The second case immediately follows from Lemma 3.

For the last case, observe, that there is a removed value  $\langle v, val' \rangle$  and all variables of the  $ker(\langle v, val' \rangle)$  except  $u$  and another variable are neutral with respect to it. Imagine that  $\langle u, val \rangle$  is appended to the current partial solution. Then all the variables of  $ker(\langle v, val' \rangle)$  will be neutral with respect to  $\langle v, val' \rangle$ . Thus we get the case number 4 of Theorem 1. ■

The worst-case complexity of procedures *propagate\_backward\_mac* and *propagate\_forward\_mac* does not exceed  $O(n^2 * m^2)$  ( $n$  is the number of variables,  $m$  is the maximal domain size).

The most subtle point of proving this is to show that a number of consecutive applications of an arc-consistency achieving algorithm does not require a greater complexity. Note that the processing of every removed value by an arc-consistency algorithm that uses supports takes  $O(n * m)$  because supports must be updated for all values and values with zero support must be added to the removing queue. Note that a CN has at most  $O(n * m)$  values, therefore complexity of an AC achieving algorithm does not exceed  $O(n^2 * m^2)$ . Observe that when the algorithm is applied a number of times during the iteration, no value can be removed twice. Therefore, the cost is amortized among all runs of the AC-algorithm during one iteration.

It is easy to see that the complexity of *propagate\_backward\_mac* is bounded by  $O(n^2 * m^2)$ , because iterative achieving of AC dominates the cost. The desired upper bound for the *propagate\_forward\_mac* procedure is ensured by the memorization technique analogous to that described for FC-RKP.

## 5. FC-RKP as a generalization of FC-CBJ

When we transformed FC-RK into FC-RKP, we added to FC-RK quite complicated pruning procedures. In this section we consider a simplified version of FC-RKP, which we term FC-RKP2. The simplified algorithm differs from FC-RKP in the following:

- it does not run the *propagate\_forward* procedure at all;
- it runs a shorter version of the *propagate\_backward* procedure which does not perform operations described in lines 4-11 of Algorithm 2.

In other words, FC-RKP2 performs additional pruning only at the backtrack stage and only if the kernel of the value being removed is empty.

In this section we prove that FC-RKP2 is identical to FC-CBJ (Prosser, 1993) with the only difference that instead of one backjump it performs a sequence of consecutive backtracks. Then we argue that FC-RKP is a generalization of FC-CBJ.

We start from brief overview of FC-CBJ. The algorithm enhances the pruning power of FC by maintaining data structures called *conflict sets*. Intuitively, the conflict set of a variable  $v$  is the set of variables whose assignments are "culprits" for removing values of  $v$ .

We denote the conflict set of  $v$  by  $conf(v)$ . Initially, the conflict sets of all variables are empty. When assignment  $\langle u, val \rangle$  is appended

to the current partial solution and the propagation procedure removes  $\langle v, val' \rangle$  because of its incompatibility with  $\langle u, val \rangle$ , the conflict set of  $v$  is updated as follows:  $conf(v) \leftarrow conf(v) \cup \{v\}$ .

The backtracking procedure of FC-CBJ is more complicated than that of FC. Like FC, FC-CBJ backtracks if the current domain of some unassigned variable  $v$  is emptied. However, it does not simply backtrack to the last assigned variable, but rather *to the last assigned variable that belongs to  $conf(v)$* . Let  $u$  be the variable FC-CBJ jumps to. Then the backtrack procedure of FC-CBJ performs the following operations:

- unassigns  $u$  and all variables that were assigned after  $u$  and removes appearances of these variables from all conflict sets;
- restores in the current domains of unassigned variables all values that are compatible with the new current partial solution;
- removes from the current domain of  $u$  its last assignment;
- updates  $conf(u)$  as follows:  $conf(u) \leftarrow conf(u) \cup conf(v) \setminus \{u\}$ .

Now we are ready to prove the main theorem of the section.

**THEOREM 4.** *Assume that FC-RKP2 and FC-CBJ are applied to the same CSP and have the same ordering heuristics of variables and values. Then their executions are identical with the only difference that instead of each backjump, FC-RKP2 performs a sequence of consecutive backtracks.*

**Proof.** To prove the theorem, we need to show the following:

- whenever FC-CBJ performs an assignment, FC-RKP2 performs the same assignment;
- whenever FC-CBJ performs a backjump, FC-RKP2 performs a sequence of consecutive backtracks that "ends" at the same variable.

The former follows from our assumption that FC-CBJ and FC-RKP2 use the same ordering heuristics for variables and values. Let us prove the latter.

The proof is by induction on the chronological sequence of backjumps generated by FC-CBJ. The following structures are relevant for a backjump:

- the value  $\langle u, val \rangle$  removed by the backjump (backjump ends at the variable  $u$  and discards the current assignment of  $u$ );
- the variable  $v$  whose empty domain caused the backjump;

- the current partial solution  $P$  at the moment when the backjump starts to execute;
- the assignments  $\langle u_1, val_1 \rangle, \dots, \langle u_k, val_k \rangle$  of the current partial solution appended after the last variable of the conflict set (these are the variables FC-CBJ jumps over them).

Consider the first backjump in the sequence. It occurs because every value of the domain of  $v$  is incompatible with some assignment of  $P$ . The last assignment of  $P$  necessarily removed some values of  $v$ . Otherwise, the domain of  $v$  would be empty *before* the last assignment. Therefore  $\langle u, val \rangle$  is the last assignment of  $P$  and FC-CBJ backjumps only one variable backwards. Observe that FC-RKP2 does the same, because both the algorithms performed the same sequence of assignments up to the dead-end.

Before we move to the induction step, let us point to an interesting connection between  $conf(v)$  and  $rs(\langle u, val \rangle)$ . Note that for the basic step  $conf(v)$  is the set of all variables whose assignments removed values of  $v$ . Denote by  $rm(\langle u, val \rangle)$  the set of variables whose assignments removed values from the variables of  $rs(\langle u, val \rangle)$ . Observe that the relation  $rm(\langle u, val \rangle) = conf(v)$  holds for the basic step because  $rs(\langle u, val \rangle) = \{v\}$ . We are going to show that the relation holds for all backjumps and this will help us prove the theorem.

Let us move to the induction step and consider a backjump which is not the first in the sequence. Denote the values of  $v$  by  $val'_1, \dots, val'_m$ . Assume that the values  $val'_1, \dots, val'_l$  are removed by incompatibility with assignments of  $P$  and the others are removed by backjumps. Then  $conf(v) = C_0 \cup C_{l+1} \cup \dots \cup C_m$ , where  $C_0$  is the set of variables whose assignments removed values of  $v$ , and  $C_i$ , for  $i$  from  $l+1$  to  $m$ , is the set contributed to  $conf(v)$  by  $\langle v, val'_i \rangle$  when it was removed.

At the same time, FC-RKP2 also performs backtrack initiated by emptying the domain of  $v$  (because all previous assignments and backtracks of both the algorithms were the same by the induction assumption). Let  $RS$  be the set formed by the *compute\_rs* procedure (that  $RS$  is the *rs*-set of some value is currently irrelevant).  $RS$  can be represented as  $\{v\} \cup rs(\langle v, val'_{l+1} \rangle) \cup \dots \cup rs(\langle v, val'_m \rangle)$ . Then the set of variables whose assignments removed values from the domains of  $RS$  is  $C_0 \cup rm(\langle v, val'_{l+1} \rangle) \cup \dots \cup rm(\langle v, val'_m \rangle)$  which by the induction assumption equals  $C_0 \cup C_{l+1} \cup \dots \cup C_m = conf(v)$ .

As far as FC-CBJ jumps directly to  $u$ , none of  $\{u_1 \dots u_k\}$  belongs to  $conf(v)$ . In other words, none of  $\langle u_1, val_1 \rangle \dots \langle u_k, val_k \rangle$  removes the values of  $RS$ . Therefore, when FC-RKP2 backtracks to  $\langle u_k, val_k \rangle$ , the corresponding *ker*-set will be empty. Analogously  $ker(\langle u_{k-1}, val_{k-1} \rangle), \dots, ker(\langle u_1, val_1 \rangle)$  will all be empty. Therefore, FC-RKP2 will perform a sequence of

consecutive backtracks until it reaches  $\langle u, val \rangle$ . As a result,  $rs(\langle u, val \rangle)$  will be set to  $RS$ . Consequently,  $rm(\langle u, val \rangle) = conf(v)$ . ■

Theorem 4 shows that responsibility sets and kernels, in essence, provide an alternative representation of conflict sets. Moreover, the representation is "richer" than conflict sets. As was shown in the proof of the theorem, conflict sets can be extracted from responsibility sets and it seems that the opposite is not true. As more informative structures, responsibility sets and kernels open up additional possibilities of pruning. If we only process the case of empty kernels, we get FC-CBJ. Alternatively, we can make appropriate propagation mechanisms for kernels of size 1, of size 2, and for larger sizes. This is in fact what FC-RKP does. Therefore, FC-RKP can be considered as a generalization of FC-CBJ.

Speaking more abstractly, Theorem 4 builds a connection between such virtually different areas of constraint reasoning as techniques of pruning "by analogy" (like symmetry breaking (Fahle et al., 2001; Focacci and Milano, 2001) or interchangeability (Choueiry and Noubir, 1998)) and the methods of intelligent backtracking (Prosser, 1993). Theorem 4 shows that all these methods use the runtime information acquired by a constraint solver and do it in a similar way.

## 6. Experimental Evaluation

To evaluate the approach, we compare MAC-RKP to MAC-CBJ.<sup>1</sup> The algorithms are applied to randomly generated CNs and to random graph coloring problems. Two measures of performance are used: the number of nodes visited and the runtime. Runtime is preferable to the number of consistency checks because the latter is more or less proportional to runtime but does not take into account additional computational overhead. Both the algorithms order variables by the Fail-First heuristic (Haralick and Elliott, 1980) which selects a variable with the smallest current domains size. The values within each variable are ordered by the min-conflict heuristic (Frost and Dechter, 1995). Every measure was obtained as average of 50 runs.

Random CNs are generated given their number of variables, domain size, density  $p_1$  and tightness  $p_2$  (Prosser, 1994). We examined four sets of instances in every one of which we fixed the former three parameters and varied the tightness over the whole  $[0, 1]$  to get problems of all

---

<sup>1</sup> The results of comparing FC-RKP to FC-CBJ look similar. We omit them in order to avoid too lengthy section of experiments.

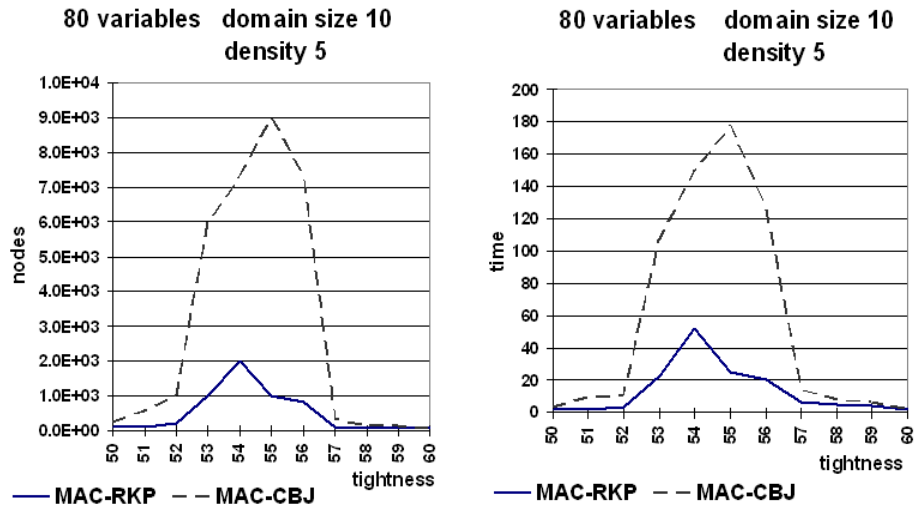


Figure 2. Number of nodes visited and run-time for random CSPs, 5% density

possible difficulties (Prosser, 1994). The resulting graphs are shown in Figures 2 - 5.

The results for every set of parameters are represented by two graphs: one on the left side, the other on the right side. The left graph compares the number of nodes visited, the right one compares the runtime.

The solid-line graphs represent the behavior of MAC-RKP, the dotted-line graphs represent the behavior of MAC-CBJ.

For every set of instances, we show the values of tightness that lie close to the phase-transition region for that set of instances. For the values of tightness that lie farther, both algorithms finish with almost no backtracks, hence their comparison is uninteresting.

It is clear that the performance of MAC-RKP is much better than that of MAC-CBJ for CNs with 5% density. For high density CNs (80%) MAC-RKP has only a slight advantage over MAC-CBJ.

The improved performance of MAC-RKP on constraint networks with low density can be explained as follows. MAC-RKP is likely to work better when the kernels generated during its run are of a small size. The size of the kernel of a value  $\langle u, val \rangle$  cannot be greater than the number of variables that have in their domains values conflicting with  $\langle u, val \rangle$ . For CNs with low density, every value is constrained with a relatively small number of variables which in turn causes small kernels produced during the execution of MAC-RKP.

It is interesting that the overhead of MAC-RKP is not large even for high density random CNs and the runtime of MAC-RKP almost never exceeds the runtime of MAC-CBJ.

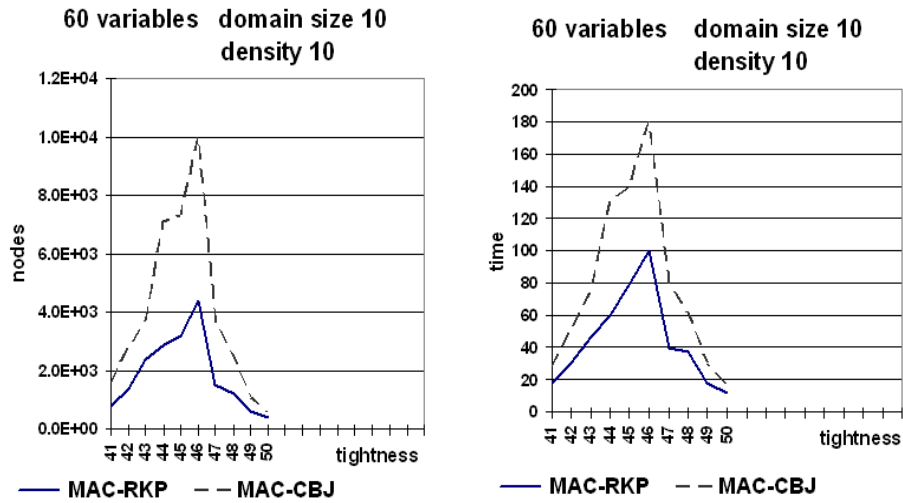


Figure 3. Number of nodes visited and run-time for random CSPs, 10% density

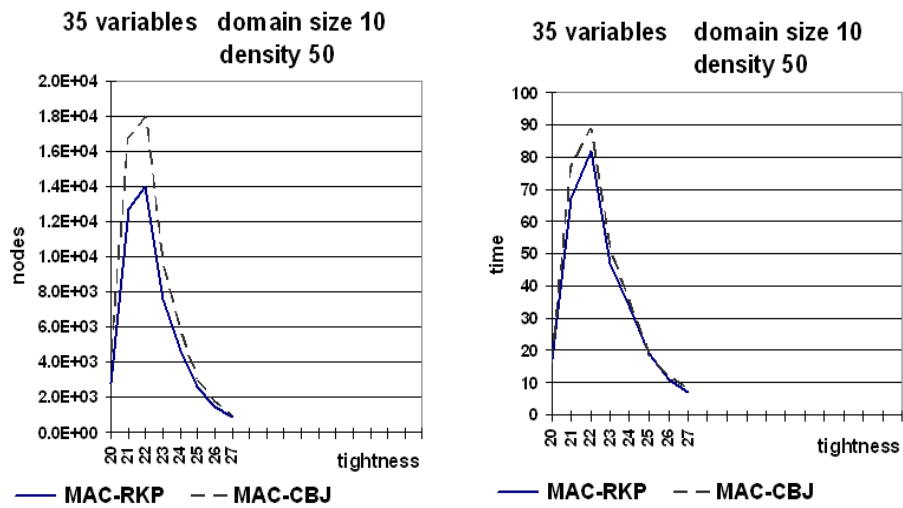


Figure 4. Number of nodes visited and run-time for random CSPs, 50% density

The proposed pruning methods and the resulting algorithms turn out to be especially efficient for graph coloring problems. We generated random graph coloring problems specifying the number of vertices, the number of colors and the density. The resulting CN has the set of variables corresponding to the set of vertices, the domain of every variable correspond to the set of colors. The pairs of variables that

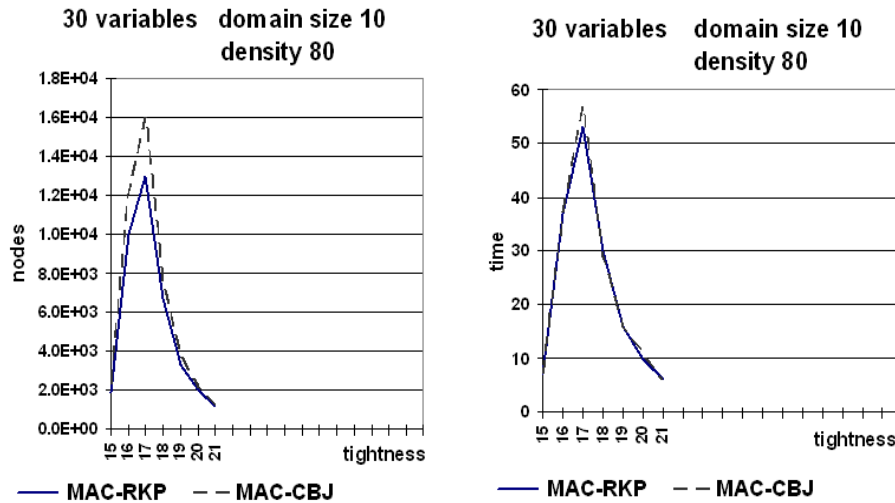


Figure 5. Number of nodes visited and run-time for random CSPs, 80% density

correspond to the pairs of vertices connected by edges are constrained by the inequality constraint.

We compared algorithms on three sets of instances. For every set of instances we fixed the number of vertices and the number of colors and varied the density. The fixed parameters are selected in such a way that for the first set of instances, the phase transition region falls in the area of small densities, for the second set of instances, phase transition occurs for medium densities. For last set, the hardest instances have a high density. The results are presented in Figures 6 - 8.

According to the experiments, the advantage of MAC-RKP over MAC-CBJ is very pronounced on graph coloring instances. MAC-RKP outperforms MAC-CBJ by a factor of 1.5 in runtime on the hard problems of low density (see Figure 6). When the region of hard graph coloring instances is of higher density, (see Figures 7, 8) the improvement factor in runtime becomes greater than 2!

## 7. Conclusions

In this paper we presented a non-resolutional approach to pruning during CSP search. This approach uses new data structures called responsibility set and kernel. We justified the proposed approach by experimental results. We also demonstrated that the presented structures have theoretical interest. They can be shown to be a generalization of

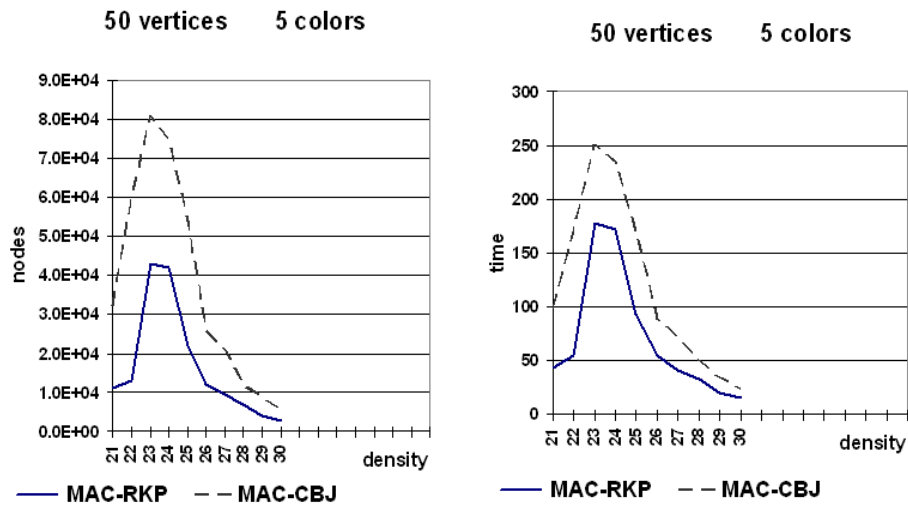


Figure 6. Hard problems of Graph coloring (low density) Nodes visited and run-time

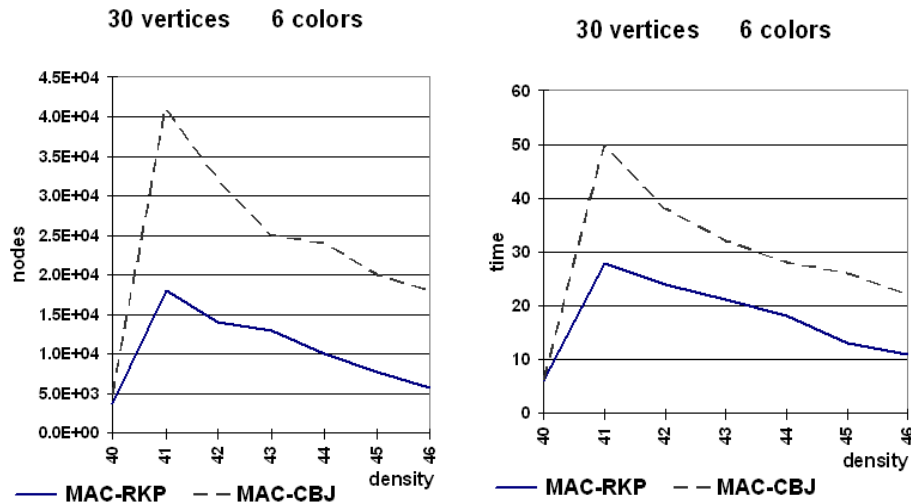


Figure 7. Hard problems of Graph coloring (medium density) Nodes visited and run-time

conflict sets and thus they connect two different areas of constraint reasoning: pruning "by analogy" and intelligent backtracking.

In this section we outline two possible directions of further research that relate to reformulation of the proposed technique to other search problems.

The proposed approach could be applied to CNs with inequality constraints using domain-dependent features of such CSPs. Two facts

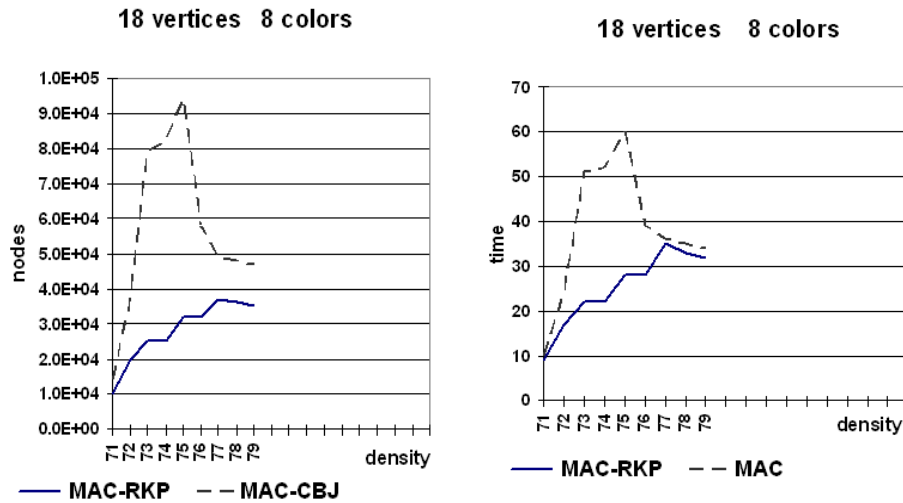


Figure 8. Graph coloring, hard problems of high density

support the evidence that a successful application is possible. First, according to our experimental results, MAC-RKP behaves well on graph-coloring problems. Second, CSPs with inequality constraints have a structural property that makes possible effective pruning using responsibility sets as shown in (Razgon and Meisels, 2004). A combination of the approaches could be particularly useful for CSPs based on inequality constraints and can be extended to some global constraints that frequently occur in resource-allocation problems. Construction of appropriate techniques based on this paradigm is a direction of our current research.

The other possible research direction is application of the proposed technique to SAT, in particular, to the methods of caching that are used in SAT solvers. Algorithms using caching, memorize unsatisfiable formulas that occur during search in order to reject the currently considered set of assignments if the residual formula induced by this set assignment is "more constrained" than one of the memorized unsatisfiable formulas. Methods of caching proved to be useful for a number of classes of SAT formulas (Kautz and Selman, 2003). It could be interesting to reformulate the notions of responsibility set and kernel in terms of SAT and to apply the notions to the methods of caching so that instead of memorizing the whole residual formulas, the new algorithms will memorize only subformulas "responsible" for their unsatisfiability.

## References

- Choueiry, B. and G. Noubir: 1998, 'On the Computation of Local Interchangeability in Discrete Constraint Satisfaction Problems'. In: *AAAI/IAAI*. pp. 326–333.
- Fahle, T., S. Schamberger, and M. Sellmann: 2001, 'Symmetry Breaking'. In: *CP2001*. pp. 93–108, Springer.
- Focacci, F. and M. Milano: 2001, 'Global Cut Framework for Removing Symmetries'. In: *CP2001*. pp. 93–108, Springer.
- Frost, D. and R. Dechter: 1995, 'Look-ahead value ordering for constraint satisfaction problems'. In: *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*. Montreal, Canada, pp. 572–578.
- Haralick, R. M. and G. Elliott: 1980, 'Increasing Tree Search Efficiency for Constraint Satisfaction Problems'. *Artificial Intelligence* **14**, 263–313.
- Kautz, H. and B. Selman: 2003, 'Ten Challenges Redux: Recent Progress in Propositional Reasoning and Search'. In: *CP2003*. pp. 1–18, Springer.
- Kondrak, G. and P. van Beek: 1995, 'A Theoretical Evaluation of Selected Backtracking Algorithms'. In: C. Mellish (ed.): *IJCAI'95*. Montreal.
- Mohr, R. and T. Henderson: 1986, 'arc and path consistency revisited'. *Artificial Intelligence* **28**, 225–233.
- Prosser, P.: 1993, 'Hybrid Algorithms for the Constraint Satisfaction Problem'. *Computational Intelligence* **9**, 268–299.
- Prosser, P.: 1994, 'Binary constraint satisfaction problems: some are harder than others'. In: *ECAI-94*. Amsterdam, pp. 95–99.
- Puget, J.: 2005, 'Symmetry Breaking Revisited'. *Constraints* **10**(1), 23–46.
- Quimper, C.-G., A. Lopez-Ortiz, P. vanBeek, and A. Golynski: 2004, 'Improved algorithms for the global cardinality constraint.'. In: *Principles and Practice of Constraint Programming-CP2004*. Toronto, Canada, pp. 542–556, Springer.
- Razgon, I. and A. Meisels: 2004, 'Pruning by Equally Constrained Variables'. In: *Proceedings of CSCLP 2004*. pp. 26–40.
- Regin, J.-C.: 1994, 'A filtering algorithm for constraints of difference in CSPs'. In: *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*. pp. 362–367, American Association for Artificial Intelligence.
- Sabin, D. and E. C. Freuder: 1994, 'Contradicting Conventional Wisdom in Constraint Satisfaction'. In: *PPCP'94*. pp. 10–20.
- Schiex, T. and G. Verfaillie: 1993, 'Two approaches to the solution maintenance problem in dynamic constraint satisfaction problems'. In: *Proc. of the IJCAI-93/SIGMAN Workshop on Knowledge-based Production Planning, Scheduling and Control, Chambery, France, (August 1993)*.