Constraints (2007) 12:151–177
DOI 10.1007/s10601-007-9016-x

A CSP Search Algorithm with Responsibility Sets and Kernels

Igor Razgon · Amnon Meisels

Published online: 28 April 2007
© Springer Science + Business Media, LLC 2007

Abstract A CSP 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 sets and kernels. We term the method that uses these structures for pruning RKP, which is abbreviation of Responsibility set, Kernel, Propagation. We combine the pruning method with algorithms FC and MAC. We call the resulting solvers FC-RKP and MAC-RKP, respectively. Experimental evaluation shows that MAC-RKP outperforms MAC-CBJ on random CSPs and on random graph coloring problems. The RKP-method also has theoretical interest. We show that under certain restrictions FC-RKP simulates FC-CBJ. It follows from the fact that intelligent backtracking implicitly uses the strategy of pruning "by analogy."

Keywords CSP search algorithm · FC-RKP · MAC-RKP · Kernel

I. Razgon (✉)
Cork Constraint Computation Centre,
University College Cork, Cork, Ireland
e-mail: i.razgon@cs.ucc.ie

A. Meisels
Department of Computer Science, Ben-Gurion University of the Negev,
Beer-Sheva 84-105, Israel
e-mail: am@cs.bgu.ac.il

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 [11, 19]. However, there are problem instances where maintaining local consistency provides little help, for example in 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 principles other 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 [15, 18] and of symmetry breaking [3, 4, 14] fall into this category.

One of the most popular symmetry breaking methods is Symmetry Breaking by Dominance Detection (SBDD) [3, 4, 14]. 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) [11] and Maintaining Arc-Consistency (MAC) [19] 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 [11] 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 2–4.5. 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 necessary background. Section 3 develops the theory related to the proposed pruning methods. Section 4 describes FC-RKP. Section 5 proves that FC-CBJ can be simulated by a restriction of FC-RKP. Section 6 presents MAC-RKP. Section 7 describes the experimental evaluation. Section 8 concludes the paper.

2 Preliminaries

2.1 Notations and Terminology

The present paper considers binary CSPs. A binary CSP 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 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.

2.2 The FC Algorithm

In this section we present the Forward Checking algorithm (FC) [8, 11]. Algorithms 1, 2, 3, and 4 present the pseudocode of FC.

In our representation, the FC algorithm maintains three auxiliary data structures: *CurSol*, *assigned*, and *validity*. *CurSol* is an ordered list of assignments. It contains the current partial solution maintained by FC. Note that although a partial solution is a set by definition, it is convenient to represent it as an ordered list in the algorithm. There are two reasons that support the ordered list representation. First, because in the backtrack stage FC performs an operation of removal of the *last* assignment of the current partial solution. Second, it will sometimes be convenient to refer to a *prefix* of the current partial solution. On the other hand, in order to prove properties of various algorithms, we frequently refer to *CurSol* as a *set* not as a *list*. This is done for convenience because otherwise we would have to use phrases like "let P be the partial solution contained in *CurSol*" which disturbs readability and makes the proofs much longer.

The Boolean array *assigned* has one entry per variable. $assigned[v] = true$ means that the variable v is assigned by the current partial solution. Otherwise, $assigned[v] = false$. We refer to the former variables as *assigned* and to the latter variables as *unassigned*.

Algorithm 1 Forward Checking

```
1:  Let Z be the CSP being solved
2:  CurSol ← ∅
3:  for every variable v of V(Z) do
4:     assigned[v] ← false
5:     for every value ⟨v, val⟩ do
6:        validity[⟨v, val⟩] = VALID
7:     end for
8:  end for
9:  R = fc_rec()
10: if R = SUCCESS then
11:    Print CurSol
12: else
13:    Print FAIL
14: end if
```

During the execution of FC, values are frequently removed from their domains and restored back. To remember which values are removed and which are not, the algorithm maintains the array *validity* with one entry per value of the underlying CSP. If $validity[\langle v, val\rangle] = VALID$, the value is not removed. For the removed values, there are two possible ways to mark the corresponding entry of *validity*. These are discussed during the description of the pseudocode. We call the set of values of a variable v that are not removed at a particular moment of execution of FC *the current domain* of v.

Algorithm 1 shows the initial part of FC. The structures *CurSol*, *assigned* and *validity* are initialized and the function $fc_rec()$ is called. We assume that the above structures are global, that is they can be read and updated by function fc_rec.

Function $fc_rec()$ (Algorithm 2) is the main component of FC because it implements the enumeration mechanism. The pseudocode can be divided into three parts. The first part (lines 1–6) processes the stopping conditions. The second part (lines 7–21) appends an assignment to the current partial solution and applies fc_rec recursively. The third part runs for the case when fc_rec fails to find a solution.

The initial part of fc_rec checks two stopping conditions (lines 1–6). If all the variables are assigned, the function propagates the $SUCCESS$ message to the upper level. If there is a variable with an empty current domain, the $FAIL$ message is propagated.

Algorithm 2 $fc_rec()$

1: **if** all variables are assigned **then**
2: Return $SUCCESS$
3: **end if**
4: **if** there is a variable v with an empty current domain **then**
5: Return $FAIL$
6: **end if**
7: Select an unassigned variable v
8: $assigned[v] \leftarrow true$
9: **while** the current domain of v is not empty **do**
10: Select a value $\langle v, val \rangle$ from the current domain of v
11: Append $\langle v, val \rangle$ to $CurSol$
12: $fc_lookahead(\langle v, val \rangle)$
13: $R \leftarrow fc_rec()$
14: **if** $R = SUCCESS$ **then**
15: Return R
16: **else**
17: $fc_restore_val()$
18: $validity[\langle v, val \rangle] \leftarrow INVALID$
19: Remove $\langle v, val \rangle$ from $CurSol$
20: **end if**
21: **end while**
22: Set $validity[\langle v, val \rangle]$ to $VALID$ for all $\langle v, val \rangle$ where $validity[\langle v, val \rangle] = INVALID$
23: $assigned[v] \leftarrow false$
24: Return $FAIL$

Algorithm 3 $fc_lookahead(\langle v, val \rangle)$

1: **for** every variable u with $assigned[u] = false$ **do**
2: **for** every value $\langle u, val' \rangle$ with $validity[\langle u, val' \rangle] = VALID$ **do**
3: **if** $\langle v, val \rangle$ is incompatible with $\langle u, val' \rangle$ **then**
4: $validity[\langle u, val' \rangle] \leftarrow |CurSol|$
5: **end if**
6: **end for**
7: **end for**

Algorithm 4 $fc_restore_val()$

1: **for** every variable u with $assigned[u] = false$ **do**
2: **for** every value $\langle u, val'\rangle$ with $validity[\langle u, val'\rangle] = |CurSol|$ **do**
3: $validity[\langle u, val'\rangle] \leftarrow VALID$
4: **end for**
5: **end for**

When fc_rec executes line 7, it is guaranteed that there are unassigned variables and the current domains of all unassigned variables are not empty. An unassigned variable v is selected in line 7, its entry in the *assigned* array is updated in line 8. The cycle in lines 9–21 scans over all values of the current domain of v. A value $\langle v, val\rangle$ is selected from the current domain of v (line 10) and appended to $CurSol$ (line 11). Then function $fc_lookahead$ (Algorithm 3) is applied. This function removes all valid values of unassigned variables that are incompatible with $\langle v, val\rangle$. Note that the function inserts to the corresponding entries of the *validity* array the length of the current partial solution. This length specifies which values has been removed because of incompatibility with $\langle v, val\rangle$.

In the final part of the cycle, fc_rec is recursively applied and further processing depends on the returned message R. If $R = SUCCESS$, this message is propagated to the upper level. Otherwise, fc_rec restores all the values that were removed because of incompatibility with $\langle v, val\rangle$ (function $fc_restore_val$ described in Algorithm 4), updates the entry of *validity* corresponding to $\langle v, val\rangle$, and removes $\langle v, val\rangle$ from $CurSol$. Further execution of fc_rec depends on the size of the current domain of v. If the size is greater than 0, another value is selected in the next iteration of the cycle. Otherwise, the function exits from the cycle, restores $VALID$ for all entries marked by $INVALID$ in the cycle and propagates the $FAIL$ message to the upper level.

Now we extend our terminology. We call the triplet $(CurSol, assigned, validity)$ *a state of FC*. Observe that the execution of FC can be represented by a sequence of states. However, not all states are of interest. For example, in a state that occurs before or during the execution of $fc_lookahead$, some entries of the *validity* array may be inconsistent because $fc_lookahead$ has not processed them yet. Therefore the states we consider include the initial state and those states that occur after assigning a variable or after backtracking.

Consider a state S, where $validity[\langle v, val\rangle] = INVALID$ (that is, $\langle v, val\rangle$ is removed by backtracking). We denote by $nogood(\langle v, val\rangle)$ the partial solution that caused discarding of $\langle v, val\rangle$. To obtain $nogood(\langle v, val\rangle)$ in the given state S, let P be the prefix of $CurSol$ that precedes the current assignment of v (if v is unassigned then $P = CurSol$). Then $nogood(\langle v, val\rangle) = P \cup \{\langle v, val\rangle\}$.

It is worth noting that the operations in lines 7 and 10 of Algorithm 1 are non-deterministic. This fact has particular importance for successful application of FC because it allows us to select an arbitrary unassigned variable and to assign it with an arbitrary value from its current domain. The heuristics that can be used for selecting a variable and a value can dramatically speed up the execution of FC. Much research has been devoted to the development of good heuristics (for example [5, 6, 20]).

3 The Pruning Technique

In this section, we develop the theory related to the main idea proposed by the present paper. The proposed pruning technique is inspired by the methods of substitutability [2] and Symmetry Breaking by Dominance Detection (SBDD) [3, 4]. According to the substitutability condition, the current partial solution maintained by FC is a nogood if the domain of the *last* variable u of the current partial solution has a value $\langle u, val_1 \rangle$ removed by backtracking which is compatible with all values of the current domains of unassigned variables.

The substitutability condition is checked only for the removed values of the *last* variable assigned by the current partial solution. In contrast, SBDD regards the removed values of *all* the variables of the current partial solution [3, 4]. A natural extension of substitutability checking is to try to do the same. However, straightforward checking of the substitutability condition for all the variables does not work. Consider, for example, the CSP shown in Fig. 1.

Consider a state where $CurSol = (\langle v_1, 2\rangle, \langle v_2, 2\rangle)$ and $\langle v_1, 1\rangle$ is removed by backtracking. In this state, the unassigned variables are v_3 and v_4. Observe that every value in the current domain of the unassigned variables is compatible with $\langle v_1, 1\rangle$. Therefore we might deduce that $CurSol$ is a nogood. However, this is not the case because appending to $CurSol$ the assignments $\langle v_3, 1\rangle$ and $\langle v_4, 1\rangle$ actually results in a full solution!

At first glance, this phenomenon seems strange because a partial solution is just a consistent set of values and two assignments of the current partial solution do not seem different. To understand the behaviour, observe that the correctness of the substitutability condition is based on the fact that in any solution containing $CurSol$ an assignment $\langle u, val\rangle \in CurSol$ can be replaced by a removed value $\langle u, val_1\rangle$. This replacement is valid only if $\langle u, val_1\rangle$ is compatible with *all* the assignments of $CurSol$ other than $\langle u, val\rangle$. This requirement automatically holds if u is the *last* variable assigned by $CurSol$: in this case $nogood(\langle u, val_1\rangle) = CurSol \setminus \{\langle u, val\rangle\} \cup \{\langle u, val_1\rangle\}$. However, this requirement does not necessarily hold if u is not the last variable. In particular, $\langle u, val_1\rangle$ can be incompatible with the assignment of any variable "later" than u. Observe that in the example above, $\langle v_1, 1\rangle$ is incompatible with $\langle v_2, 2\rangle$. Thus

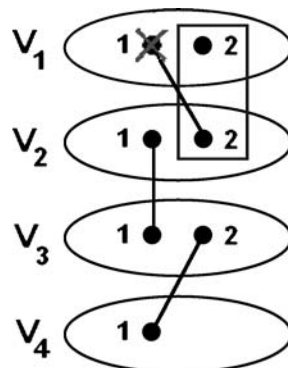

Fig. 1 Incorrect approach of generalization of substitutability checking

we may conclude that the additional requirement for the substitutability condition is that all assignments inserted into *CurSol* after $\langle u, val \rangle$ are compatible with $\langle u, val_1 \rangle$.

It is convenient to describe the generalized substitutability condition using the following definition of a neutral variable.

Definition 1 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 $\langle u, val \rangle$.

Now, the generalized substitutability condition can be formulated as follows [16]:

Proposition 1 *Consider a state that occurs during execution of FC. Assume that there is a value $\langle u, val_1 \rangle$ removed by backtracking such that all the variables that are not assigned by nogood($\langle u, val_1 \rangle$) are neutral with respect to $\langle u, val_1 \rangle$. Then CurSol is a nogood.*

Proof Assume by contradiction that the statement is not true and let P be a solution of the underlying CSP that contains *CurSol*. Also, let $\langle u, val \rangle$ be the assignment of u in *CurSol*, and let P_1, P_2 and P_3 be disjoint subsets of P, where P_1 is the subset of *CurSol* placed before $\langle u, val \rangle$, P_2 is the subset of *CurSol* placed after $\langle u, val \rangle$, and P_3 assigns the variables unassigned by *CurSol*. Clearly, $P = P_1 \cup \{\langle u, val \rangle\} \cup P_2 \cup P_3$.

Observe that $\langle u, val_1 \rangle$ is compatible with P_1 only because $nogood(\langle u, val_1 \rangle) = P_1 \cup \langle u, val_1 \rangle$. Also, $\langle u, val_1 \rangle$ is compatible with P_2 and P_3 because the variables assigned by these partial solutions are neutral with respect to $\langle u, val_1 \rangle$. Thus, if we replace in P $\langle u, val \rangle$ by $\langle u, val_1 \rangle$, we will get another solution of the underlying CSP. However, this new solution will contain $nogood(\langle u, val_1 \rangle)$, which contradicts the definition of a nogood. □

We demonstrate the application of the generalized substitutability condition on the CSP shown in Fig. 2. Consider the state of FC where $CurSol = (\langle v_1, 2 \rangle, \langle v_2, 1 \rangle)$ and $\langle v_1, 1 \rangle$ has been removed by backtracking. Observe that variables v_2, v_3, v_4, and v_5 are neutral with respect to $\langle v_1, 1 \rangle$ Then, according to Proposition 1, *CurSol* is a nogood, which is easy to verify.

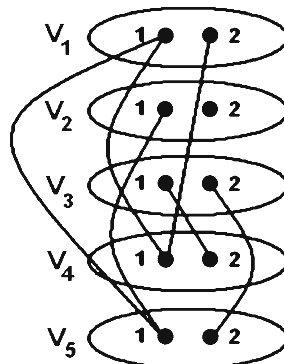

Fig. 2 Generalized substitutability checking

We observe that to discard the current partial solution it is not necessary that *all* the variables considered in Proposition 1 be neutral with respect to $\langle u, val_1\rangle$. This is presented formally by introducing the notion of a responsibility set.

Definition 2 Let P be a nogood of a CSP 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 example, in the CSP shown in Fig. 2, a responsibility set of $\{\langle v_1, 1\rangle\}$ is $\{v_3, v_4, v_5\}$.

Assume that the search algorithm maintains an array *resp* whose entries correspond to the values of the underlying CSP. For a value $\langle u, val\rangle$ removed by backtracking, $resp[\langle u, val\rangle]$ is a responsibility set of $nogood(\langle u, val\rangle)$. Given the *resp*-array the following pruning condition can be stated.

Proposition 2 *Consider a state that occurs during execution of FC. Assume that there is a value $\langle u, val_1\rangle$ removed by backtracking such that all the variables of $resp[\langle u, val_1\rangle]$ are neutral with respect to $\langle u, val_1\rangle$. Then CurSol is a nogood with a responsibility set $resp[\langle u, val_1\rangle] \setminus A$, where A is the subset of $resp[\langle u, val_1\rangle]$ assigned by CurSol.*

Proof Assume by contradiction that there is a partial solution P that contains *CurSol* and assigns all the variables of $resp[\langle u, val_1\rangle]$.

Let $\langle u, val\rangle$ be the assignment of u in P. Then, analogously to the proof of Proposition 1, we distinguish three subsets of P: P_1 containing assignments appended to *CurSol* before $\langle u, val\rangle$; P_2 containing assignments appended to *CurSol* after $\langle u, val\rangle$; P_3 assigning variables that are not assigned by *CurSol*.

Analogously to the proof of Proposition 1, we can observe that $\langle u, val_1\rangle$ is compatible with P_1, P_2, and P_3. Then, replacing $\langle u, val\rangle$ by $\langle u, val_1\rangle$ we get another partial solution, that contains $nogood(\langle u, val_1\rangle)$ and assigns all the variables of $resp(\langle u, val_1\rangle)$ in contradiction to the definition of a responsibility set.

Let $V(P_3)$ be the set of variables assigned by P_3 and observe that $CurSol = P_1 \cup \{\langle v_1, val\rangle\} \cup P_2$. Taking into account that P_3 is an arbitrary partial solution Consistent with *CurSol*, it follows that any consistent extension Of *CurSol* to the variables of $V(P_3)$ is a nogood. That is $V(P_3)$ is a responsibility set of *CurSol*. By definition, $V(P_3)$ includes all the variables of $resp[\langle v_1, val\rangle]$ that are *not* assigned by *CurSol*. Denote by A the variables of $resp[\langle v_1, val\rangle]$ assigned by *CurSol*. It follows that $V(P_3) = resp[\langle u, val_1\rangle] \setminus A$. □

To demonstrate the application of Proposition 2, consider the CSP shown in Fig. 3. Note that the CSP there is almost the same as that of Fig. 2. The differences are that variables v_4, v_5 are called v_{100}, v_{200}, respectively, and that we assume that the CSP has many other variables. Similarly to the previous example, we assume that $CurSol = (\langle v_1, 2\rangle, \langle v_2, 1\rangle)$ and that $\langle v_1, 1\rangle$ has been removed by backtrack. Observe that $\{v_3, v_{100}, v_{200}\}$ is a responsibility set of $nogood(\langle v_1, 1\rangle) = \{\langle v_1, 1\rangle\}$. Therefore, the neutrality of these variables with respect to $\langle v_1, 1\rangle$ is sufficient to deduce that *CurSol* is a nogood.

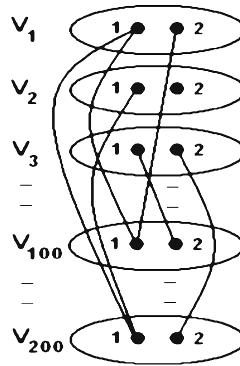

Fig. 3 Generalized substitutability checking

Checking neutrality only for variables of $resp(\langle u, val_1 \rangle)$ has the following two advantages.

– The neutrality checking of an unassigned variable is a relatively heavy operation. The set $resp[\langle u, val_1 \rangle]$ may include only a small subset of all the unassigned variables, which essentially speeds up the checking procedure.
– It is more likely that the pruning condition will be satisfied.

We can further speed up the pruning procedure since it is not necessary to check explicitly the neutrality of all variables of $resp[\langle u, val_1 \rangle]$. In our last example (Fig. 3), we can see that it makes no sense to check the neutrality of v_3 with respect to $\langle v_1, 1 \rangle$ because as a result of appending $\langle v_1, 1 \rangle$ to the current partial solution, no value is removed from the current domain of v_3.

We suggest that it is sufficient to check neutrality only for the set of variables whose domains have values incompatible with $\langle u, val_1 \rangle$ and compatible with the other assignments of $nogood(\langle u, val_1 \rangle)$. We provide a formal definition of this set.

Definition 3 Let P be a nogood of a CSP 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 $K \subseteq S$ of all the variables of S whose domains have values that are incompatible with $\langle u, val \rangle$, but are compatible with the other values of P.

Returning again to our last example, we see that the kernel of $\langle v_1, 1 \rangle$ with respect to $nogood(\langle v_1, 1 \rangle)$ and $\{v_3, v_{100}, v_{200}\}$ is $\{v_{100}, v_{200}\}$.

Assume that FC is modified so that it associates with every value $\langle u, val_1 \rangle$ removed by backtracking a set $ker[\langle u, val_1 \rangle]$ which is the kernel of $\langle u, val_1 \rangle$ with respect to $nogood(\langle u, val_1 \rangle)$ and $resp[\langle u, val_1 \rangle]$. Then Proposition 2 can be reformulated as follows.

Proposition 3 *Consider a state that occurs during execution of FC. Assume that there is a value $\langle u, val_1 \rangle$ removed by backtracking such that all the variables of $ker[\langle u, val_1 \rangle]$ are neutral with respect to $\langle u, val_1 \rangle$. Then CurSol is a nogood with a responsibility set $resp[\langle u, val_1 \rangle] \setminus A$, where A is the subset of $resp[\langle u, val_1 \rangle]$ assigned by CurSol.*

Proof The present proposition will directly follow from Proposition 2 if we show that every variable of $resp[\langle u, val_1 \rangle] \setminus ker[\langle u, val_1 \rangle]$ is neutral with respect to $\langle u, val_1 \rangle$.

Let $w \in resp[\langle u, val_1\rangle] \setminus ker[\langle u, val_1\rangle]$. We shall show that all values of the current domain of w are compatible with $\langle u, val_1\rangle$ (observe that in this case w is neutral with respect to $\langle u, val_1\rangle$ even if it is assigned).

Let $\langle u, val\rangle$ be the assignment of u in P, and let P_1 be the subset of $CurSol$ preceding $\langle u, val\rangle$. Note that $nogood(\langle u, val_1\rangle) = P_1 \cup \{\langle u, val_1\rangle\}$. By the definition of $resp[\langle u, val_1\rangle]$, w is not assigned by P_1. Taking that into account and considering that P_1 is a subset of the current partial solution, we get that all the values of the current domain of w are compatible with P_1. It follows that, if some value of the current domain of w is incompatible with $\langle u, val_1\rangle$ then w must belong to $ker[\langle u, val_1\rangle]$ in contradiction to the selection of w. □

The condition in Proposition 3 allows much quicker checking than the condition suggested by Proposition 2 because $|ker(\langle u, val_1\rangle)|$ is at most the number of variables that have values incompatible with $\langle u, val_1\rangle$, while $resp(\langle u, val_1\rangle)$ might have an arbitrary size. The improvement in runtime is especially pronounced for CSPs of low density.

4 The FC-RKP Algorithm

4.1 Algorithm Description

In this section we describe a modification of FC that performs additional pruning according to the techniques described in the previous section. We call the resulting algorithm FC-RKP (RKP is the abbreviation of Responsibility sets, Kernels and Pruning). The pseudocode is presented in Algorithms 5, 6, 7, and 8. The former three algorithms are based on Algorithms 1, 2, and 3 but perform a number of additional operations. To emphasize the differences, the operations or structures that do not appear in Algorithms 1, 2, and 3 are emphasized by boxes.

Algorithm 5 describes the initialization procedure of FC-RKP.

Algorithm 5 FC-RKP

```
1:  Let Z be the CSP being solved
2:  CurSol ← ∅
3:  for every variable v of V(Z) do
4:     assigned[v] ← false
5:     for every value ⟨v, val⟩ do
6:        validity[⟨v, val⟩] = VALID
7:        [ resp[⟨v, val⟩] = ∅ ]
8:     end for
9:  end for
10: R = fc_rkp_rec()
11: if R = SUCCESS then
12:    Print CurSol
13: else
14:    Print FAIL
15: end if
```

Algorithm 6 $fc_rkp_rec()$

1: **if** all variables are assigned **then**
2: Return $SUCCESS$
3: **end if**
4: **if** there is a variable v with the empty current domain **then**
5: Return $\boxed{(FAIL, UR(v))}$
6: **end if**
7: $\boxed{R \leftarrow fc_rkp_check_neutrality()}$
8: **if** $\boxed{R \neq SUCCESS}$ **then**
9: $\boxed{\text{Return } R}$
10: **end if**
11: Select an unassigned variable v
12: $assigned[v] \leftarrow true$
13: **while** the current domain of v is not empty **do**
14: Select a value $\langle v, val\rangle$ From the current domain of v
15: Append $\langle v, val\rangle$ to $CurSol$
16: $\boxed{ker[\langle v, val\rangle]} \leftarrow fc_rkp_lookahead(\langle v, val\rangle)$
17: $R \leftarrow fc_rkp_rec()$
18: **if** $R = SUCCESS$ **then**
19: Return R
20: **else**
21: $fc_restore_val()$
22: $validity[\langle v, val\rangle] \leftarrow INVALID$
23: Remove $\langle v, val\rangle$ from $CurSol$
24: $\boxed{\text{Let } resp[\langle v, val\rangle] \text{ be the second component of } R}$
25: $\boxed{ker[\langle v, val\rangle] \leftarrow ker[\langle v, val\rangle] \cap resp[\langle v, val\rangle]}$
26: **if** $\boxed{ker[\langle v, val\rangle] = \emptyset}$ **then**
27: $\boxed{\text{Set } resp[\langle v, val\rangle] \text{ to } \emptyset \text{ for all } \langle v, val\rangle \text{ with } validity[\langle v, val\rangle] = INVALID}$
28: $\boxed{\text{Set } validity[\langle v, val\rangle] \text{ to } VALID \text{ for all } \langle v, val\rangle \text{ with } validity[\langle v, val\rangle] = INVALID}$
29: $\boxed{assigned[v] \leftarrow false}$
30: $\boxed{\text{Return } R}$
31: **end if**
32: **end if**
33: **end while**
34: $\boxed{\text{Set } resp[\langle v, val\rangle] \text{ to } \emptyset \text{ for all } \langle v, val\rangle \text{ with } validity[\langle v, val\rangle] = INVALID}$
35: Set $validity[\langle v, val\rangle]$ to $VALID$ for all $\langle v, val\rangle$ with $validity[\langle v, val\rangle] = INVALID$
36: $assigned[v] \leftarrow false$
37: Return $\boxed{(FAIL, UR(v))}$

Algorithm 6 describes procedure fc_rkp_rec, the main engine of FC-RKP. We describe in detail the differences of fc_rkp_rec from fc_rec. First of all, note that the $FAIL$ message returned by fc_rkp_rec consists of two components: the $FAIL$ token itself and the subset $UR(v)$ of variables that consists of the variable v whose empty domain caused the $FAIL$ message and the union of *resp* entries of all the values of v (the message is explicitly constructed in lines 5 and 37).

In lines 7–10, the $FAIL$ message is returned if the pruning condition stated in Proposition prop:ker is satisfied. The pruning condition is checked in line 8 by applying the function $fc_rkp_check_neutrality$. Algorithm 8 presents pseudocode of the function. If the condition is not satisfied, fc_rkp_rec performs an iteration of variable instantiation.

A variable to be assigned is selected in lines 11–12 of fc_rkp_rec. Lines 14–17 list the actions related to assigning a variable. Note that the function of lookahead applied in line 17 of Algorithm 6 and described in Algorithm 7 returns the set of variables whose values have been removed. This set initializes $ker[\langle v, val\rangle]$. The final content of the entry is computed when the assignment is discarded. The function fc_rkp_rec is recursively applied in line 18. Further execution depends on the answer it returns. If the answer is $SUCCESS$, it is propagated to the upper level. Otherwise, fc_rk_rec starts the process of backtracking.

The backtracking process is described in lines 21–31. Lines 21–23 are analogous to the backtracking operations of fc_rec (Algorithm 2). The set $resp[\langle v, val\rangle]$ is computed in line 24. The kernel of $\langle v, val\rangle$ is computed in line 25. Recall that the kernel is a subset of $resp[\langle v, val\rangle]$ that contains variables having in their current domains values incompatible with $\langle v, val\rangle$. However, the incompatible values are identified by the lookahead function, so the straightforward computation of the kernel is redundant. Instead, $ker[\langle v, val\rangle]$ is initialized in line 16 to the set of all variables whose current domains have values incompatible with $\langle v, val\rangle$. What remains to be done in line 25 is to find the intersection of this set with $resp[\langle v, val\rangle]$.

In lines 26–31, fc_rkp_rec checks whether $ker[\langle v, val\rangle]$ is empty. If it is, an additional backtrack is executed. Note that this mechanism resembles FC-CBJ. Further in the paper we show the validity of this intuition by proving that FC-RKP is a generalization of FC-CBJ.

If the variable v fails to be assigned then fc_rkp_rec executes a backtrack (lines 34–37).

Algorithm 7 $fc_rkp_lookahead(\langle v, val\rangle)$

```
1:  [kernel ← ∅]
2:  for every variable u with assigned[u] = false do
3:     for every value ⟨u, val′⟩ with validity[⟨u, val′⟩] = VALID do
4:        if ⟨v, val⟩ is incompatible with ⟨u, val′⟩ then
5:           validity[⟨u, val′⟩] ← |CurSol|
6:           [kernel ← kernel ∪ {u}]
7:        end if
8:     end for
9:  end for
10: [Return kernel]
```

Algorithm 8 $fc_rkp_check_neutrality()$

1: **for** every value $\langle u, val \rangle$ such that $validity[\langle u, val \rangle] = INVALID$ **do**
2: **if** all variables of $ker(\langle u, val \rangle)$ are neutral with respect to $\langle u, val \rangle$ **then**
3: Let A be the subset of $resp(\langle u, val \rangle)$ assigned by $CurSol$
4: Return $(FAIL, resp(\langle u, val \rangle) \setminus A)$
5: **else**
6: Return $SUCCESS$
7: **end if**
8: **end for**

4.2 Correctness Proof

To prove the correctness of FC-RKP, one must prove soundness, termination, and completeness. The soundness and termination of FC-RKP follow from the soundness and termination of FC. It remains to prove completeness, that is, to show that removal of values by FC-RKP does not cause a loss of a solution.

The completeness of FC-RKP is based on the following two lemmas.

Lemma 1 *When FC-RKP removes a value* $\langle u, val \rangle$ *because of its incompatibility with the current partial solution,* $resp[\langle u, val \rangle]$ *is set to* $\emptyset$.

Proof Assume that $\langle u, val \rangle$ is removed because of incompatibility with the last value of $CurSol$. Observe that $resp[\langle u, val \rangle]$ is not changed when $\langle u, val \rangle$ is removed. Thus, to prove that $resp[\langle u, val \rangle] = \emptyset$, it is enough to show that the entry $resp[\langle u, val \rangle]$ contained $\emptyset$ before the removal of $\langle u, val \rangle$.

To show this, note, that initially $resp[\langle u, val \rangle] = \emptyset$. The content of $resp[\langle u, val \rangle]$ can change only if $\langle u, val \rangle$ is removed by backtracking. However, when the value is restored to the current domain, $resp[\langle u, val \rangle]$ is reinitialized to $\emptyset$. □

Lemma 2 *Every time when FC-RKP removes a value* $\langle u, val \rangle$ *by backtracking, the set recorded in* $resp[\langle u, val \rangle]$ *is a responsibility set of* $nogood[\langle u, val \rangle]$.

Proof Recall that the backtracking process is initiated by a $FAIL$ message returned by the recursive call in line 17 of Algorithm 6. The second component of the message is recorded by FC-RKP in $resp[\langle u, val \rangle]$. Note that for the recursive call of fc_rkp_rec that performs backtracking, $nogood(\langle u, val \rangle)$ is just $CurSol$. Therefore, it is sufficient to show that every time when fc_rkp_rec returns $FAIL$, the second component of the $FAIL$ message is the responsibility set of $CurSol$.

There are four possible reasons why fc_rkp_rec returns $FAIL$.

1. An unassigned variable with an empty current domain is detected (lines 4–6 of Algorithm 6).
2. The function $fc_rkp_check_neutrality$ returns the $FAIL$ message (lines 7–10 of Algorithm 6).
3. The kernel of the value removed in the last backtrack is empty (lines 26–31 of Algorithm 6).

4. An attempt to assign a variable in the main cycle (lines 13–33) has failed. In this case the $FAIL$ message is returned in lines 34–37.

Let us consider a chronological sequence of states that cause generation of the $FAIL$ message and prove the present lemma by induction on this sequence.

Consider the state when fc_rkp_rec returns $FAIL$ for the first time. There are no values removed by backtracks yet, hence the only reason that caused the function to return $FAIL$ is an empty current domain of some unassigned variable v. All values of v are removed because of incompatibility with $CurSol$, that is, their corresponding *resp*-entries are empty (Lemma 1). Therefore the second component of the $FAIL$ message is $\{v\}$. Observe that $\{v\}$ is indeed a responsibility set of $CurSol$. Thus the lemma holds for the first state of the considered sequence.

Consider the state which is not first in the analyzed sequence and assume that the theorem holds for all the previous states.

In this state, the $FAIL$ message can be caused by any of the four reasons stated previously. Assume it is caused by the first or by the last ones. This would mean that there is an unassigned variable v with an empty current domain (the values of v can be removed either by incompatibility or by backtracking). Let S be the second component of the $FAIL$ message returned by fc_rkp_rec. Let us show that S is the responsibility set of $CurSol$.

If all values of v are incompatible with $CurSol$, then the situation is analogous to that considered for the first step of the induction.

If the domain of v includes values compatible with $CurSol$, we let $\langle v, val' \rangle$ be such a value. Observe that $\langle v, val' \rangle$ has been removed by backtracking and that $nogood(\langle v, val' \rangle) = CurSol \cup \{\langle v, val' \rangle\}$. By the induction assumption, $resp[\langle v, val' \rangle]$ is a responsibility set of $nogood(\langle v, val' \rangle)$. Observe that $resp[\langle v, val' \rangle] \subseteq S \setminus \{v\}$ by definition of S (see line 37 of Algorithm 6). It follows that $S \setminus \{v\}$ is a responsibility set of $nogood(\langle v, val' \rangle)$. Thus, the extension of $CurSol$ by appending any compatible value of v, produces a nogood with a responsibility set $S \setminus \{v\}$. Consequently, $CurSol$ itself is a nogood with a responsibility set S.

If a return of $FAIL$ was caused by the second reason then there is a value $\langle w, val' \rangle$ removed by backtracking such that all variables of $ker[\langle w, val' \rangle]$ are neutral with respect to $\langle w, val' \rangle$. The present lemma holds in this case according to the induction assumption and Proposition 3. The third reason is just a degenerated case of the second reason, so the validity of the lemma follows again from Proposition 3. □

Theorem 1 *FC-RKP is complete.*

Proof According to Lemma 2, $CurSol$ is a nogood whenever its LAST assignment is discarded. The completeness of FC-RKP follows immediately from this fact. □

5 Responsibility Sets as an Alternative Representation of Conflict Sets and Eliminating Explanations

In this section we show that responsibility sets are at least as informative as conflict sets by proving that the FC-CBJ algorithm [11] can be simulated by using responsibility sets and kernels. Then we briefly argue that algorithms based on eliminating explanations [7, 9] can be also represented by responsibility sets by showing that

an eliminating explanation of a value can be extracted from a responsibility set associated with that value.

We start with a 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 to 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: $conf(u) \leftarrow conf(u) \cup conf(v) \setminus \{u\}$.

Consider a modification of FC-RKP that does not execute lines 7–10 of Algorithm 6. That is, the only additional pruning it performs is initiating a sequence of backtracks until the *ker*-entry of the removed value is not empty. We call this modification FC-RKP2. The following theorem shows that FC-RKP2 is analogous to FC-CBJ.

Theorem 2 *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 $\langle u_1, val_1\rangle, \ldots, \langle u_k, val_k\rangle$ of the current partial solution appended after the last variable of the conflict set (these are the variables FC-CBJ jumps over).

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 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 $resp[\langle u, val\rangle]$. 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 $resp[\langle u, val\rangle]$. Observe that the relation $rm(\langle u, val\rangle) = conf(v)$ holds for the basic step because $resp[\langle u, val\rangle] = \{v\}$. 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 $val'_1, \ldots val'_m$. Assume that the values $val'_1, \ldots val'_l$ are removed by incompatibility with assignments of P and the others are removed by backjumps. Then $conf(v) = C_0 \cup C_{l+1} \cup \cdots \cup C_m$, 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 a backtrack initiated by emptying the domain of v (because all previous assignments and backtracks of both algorithms were the same by the induction assumption). Let RS be the set returned by fc_rkp_rek in line 37. RS can be represented as $\{v\} \cup resp[\langle v, val'_{l+1}\rangle] \cup \cdots \cup resp[\langle v, val'_m\rangle]$. Then the set of variables whose assignments removed values from the domains of RS is $C_0 \cup rm(\langle v, val'_{l+1}\rangle) \cup \cdots \cup rm(\langle v, val'_m\rangle)$ which by the induction assumption equals $C_0 \cup C_{l+1} \cup \cdots \cup C_m = conf(v)$.

As far as FC-CBJ jumps directly to u, none of $\{u_1 \ldots u_k\}$ belongs to $conf(v)$. In other words, none of $\langle u_1, val_1\rangle \ldots \langle u_k, val_k\rangle$ removes the values of RS. Therefore, when FC-RKP2 backtracks to $\langle u_k, val_k\rangle$, the corresponding *ker*-set will be empty. For the same reason, $ker(\langle u_{k-1}, val_{k-1}\rangle), \ldots, ker(\langle u_1, val_1\rangle)$ will all be empty. Therefore, *FC-RKP2 will perform a sequence of consecutive backtracks until it reaches* $\langle u, val\rangle$. As a result, $resp[\langle u, val\rangle]$ will be set to RS. Consequently, $rm(\langle u, val\rangle) = conf(v)$. □

Theorem 2 shows that responsibility sets and kernels provide an alternative representation of conflict sets. This representation is at least as informative as conflict sets: as was shown in the proof of the theorem, conflict sets can be extracted from responsibility sets. Responsibility sets and kernels also open up additional possibilities of pruning. If we only process the case of empty kernels, we obtain FC-CBJ. Alternatively, we can create 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.

Furthermore, we believe that responsibility sets have enough power to represent eliminating explanations [7, 9]. Consider the set $rm(\langle u, val \rangle)$ utilized in the proof of Theorem 2 and let P' be the subset of the current partial solution that assigns the variables of $rm(\langle u, val \rangle)$. By definition, P' contains the only assignments that remove values from $resp[\langle u, val \rangle]$. It follows that P' is a nogood. Consequently, $P' \setminus \langle u, val \rangle$ is an *eliminating explanation* of $\langle u, val \rangle$. We conjecture that with the above representation, one can simulate algorithms like MAC-DBT [9] and CCFC—[1][1] by introducing minor modifications to FC-RKP.

Speaking more abstractly, Theorem 2 together with the subsequent argumentation draw a connection between such virtually different areas of constraint reasoning as techniques of pruning "by analogy," like symmetry breaking [3, 4] or substitutability [2], and methods of intelligent backtracking [1, 7, 9, 11]. It follows from the above discussion that all of these methods use the information acquired by a constraint solver during execution in a similar manner.

6 The MAC-RKP Algorithm

6.1 Description of the Algorithm

The MAC-RKP algorithm is a combination of Maintaining Arc-Consistency (MAC) [19] and the pruning technique that uses responsibility sets and kernels. We describe MAC-RKP by explaining the modifications introduced to FC-RKP in order to obtain MAC-RKP.

First of all, MAC-RKP, as any algorithm using MAC, must achieve Arc-Consistency (AC) at the preprocessing stage. Next, the appropriate lookahead procedure must be applied. In particular, instead of the *fc_rkp_lookahead* function, the *maintain_ac_rkp* (Algorithm 9) function is applied.

The next step is to design a function of restoring values removed during lookahead. For this purpose, the function *fc_restore_val* used in FC-RKP is slightly modified. That is, we set the *resp*-entries of values restored by the function to $\emptyset$.

The last modification for constructing MAC-RKP is to design a function for checking neutrality. As for FC-RKP, the function decides that the current partial solution is a nogood if there is a value $\langle u, val \rangle$ removed by backtracking such that every variable of $ker[\langle u, val \rangle]$ is neutral with respect to $\langle u, val \rangle$. An important difference from FC-RKP is the method of computing a responsibility set accompanying the $FAIL$ message. The resulting responsibility set must include the *resp*-entries of all values incompatible with $\langle u, val \rangle$ that belong to the domains of variables of $ker[\langle u, val \rangle]$. This mechanism of computing responsibility sets is implemented by the function *mac_rkp_check_neutrality* (Algorithm 10). MAC-RKP uses this function instead of *fc_rkp_check_neutrality*.

In this function every time a new value $\langle u, val \rangle$ removed by backtracking is checked, the variable RS is initialized to $resp[\langle u, val \rangle]$. Then every time the function

[1] The structures associated with removed values in algorithms CCFC and CCFC—are not originally called eliminating explanations, but it is not hard to show their equivalence.

Algorithm 9 *maintain_ac_rkp*($\langle v, val\rangle$)

1: $ac_queue \leftarrow \emptyset$
2: $kernel \leftarrow \emptyset$
3: **for** every variable u with $assigned[u] = false$ **do**
4:   **for** every value $\langle u, val'\rangle$ with $validity[\langle u, val'\rangle] = VALID$ **do**
5:     **if** $\langle v, val\rangle$ is incompatible with $\langle u, val'\rangle$ **then**
6:       $validity[\langle u, val'\rangle] \leftarrow |CurSol|$
7:       $enqueue(ac_queue, \langle u, val'\rangle)$
8:       $kernel \leftarrow kernel \cup \{u\}$
9:     **end if**
10:   **end for**
11: **end for**
12: **while** ac_queue is not empty **do**
13:   $\langle u, val'\rangle = dequeue(ac_queue)$
14:   **for** every variable w except u **do**
15:     **for** every value $\langle w, val''\rangle$ **do**
16:       **if** $\langle w, val''\rangle$ is incompatible with all the values of the current domain of u **then**
17:         $validity[\langle w, val''\rangle] \leftarrow |CurSol|$
18:         Let UR be the union of *resp* entries of all values of u
19:         $resp[\langle w, val''\rangle] = UR \cup \{u\}$
20:         $enqueue(ac_queue, \langle w, val''\rangle)$
21:       **end if**
22:     **end for**
23:   **end for**
24: **end while**
25: Return *kernel*

detects a variable $w \in ker[\langle u, val\rangle]$, it adds to RS the union of *resp*-entries of all values of w that are incompatible with $\langle u, val\rangle$. If the function returns the $FAIL$ message, the obtained set without assigned variables is returned as the second component of the message.

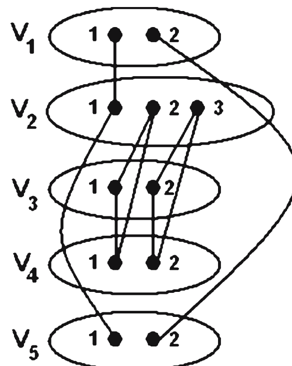

Fig. 4 A CSP on which computing of responsibility sets for MAC-RKP is demonstrated

Algorithm 10 *mac_rkp_check_neutrality*()

1: **for** every value $\langle u, val\rangle$ such that $validity[\langle u, val\rangle] = INVALID$ **do**
2: $\quad RS \leftarrow resp[\langle u, val\rangle]$
3: $\quad$ **for** every variable w of $ker[\langle u, val\rangle]$ **do**
4: $\quad\quad$ **if** w is neutral with respect to $\langle u, val\rangle$ **then**
5: $\quad\quad\quad$ Let UR be the union of *resp* entries of all values of w that are incompatible with $\langle u, val\rangle$
6: $\quad\quad\quad RS \leftarrow RS \cup UR$
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ Goto line 14
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad$ Let A be the subset of RS assigned by $CurSol$
12: $\quad$ Return $(FAIL, RS \setminus A)$
13: **end for**
14: Return $SUCCESS$

At first glance, the updating of RS executed in line 6 of Algorithm 10 seems redundant. However, this is actually necessary for preserving consistency. Let us consider the following example.

Assume that the CSP shown in Fig. 4 is solved by MAC-RKP. Also assume that $\langle v_1, 1\rangle$ has been removed and associated with a *resp*-entry $\{v_2, v_3, v_4\}$, and that the current partial solution is $\{\langle v_1, 2\rangle\}$. Then $\langle v_5, 2\rangle$ is removed because of incompatibility with $\langle v_1, 2\rangle$, and $\langle v_2, 1\rangle$ is removed after that because it is incompatible with the current domain of v_5 (remember that AC is achieved during lookahead). The entry $resp[\langle v_2, 1\rangle]$ is set to $\{v_5\}$. After removing $\langle v_2, 1\rangle$, the variables v_2, v_3, and v_4 become neutral with respect to $\langle v_1, 1\rangle$. That is $\langle v_1, 2\rangle$ can be removed. However, it is not correct to set $resp[\langle v_1, 2] = \{v_2, v_3, v_4\}$ because it is *not* a responsibility set of $\{\langle v_2, 1\rangle\}$; this is clear when we consider, for example, a partial solution $\{\langle v_1, 2\rangle, \langle v_2, 1\rangle, \langle v_3, 1\rangle, \langle v_4, 2\rangle\}$. Thus v_5 must be added to $resp[\langle v_1, 2\rangle]$.

6.2 Proof of Completeness of MAC-RKP

The completeness of MAC-RKP can be proven analogously to the completeness of FC-RKP. The main task is to show that whenever a value $\langle u, val\rangle$ compatible with $CurSol$ is removed, $resp[\langle u, val\rangle]$ is a responsibility set of $nogood(\langle u, val\rangle)$. The same inductive form of proof is used. That is, we consider the sequence of states that cause generation of the $FAIL$ message and prove the above statement by induction on the sequence. However, to prove completeness of MAC-RKP, one also has to consider those cases of removal that cannot occur during the execution of FC-RKP. The rest of the section concentrates on these cases only. In particular, we prove two propositions; the first considers removal of a value during maintaining of AC, the second proposition proves correctness of a responsibility set returned by the *mac_rkp_check_neutrality* function.

Proposition 4 *Consider a state that occurs during execution of MAC-RKP. Let u and v be two unassigned variables. Assume that a value* $\langle u, val\rangle$ *of the current domain of u is in conflict with all the values of the current domain of v. Assume also that for every removed value* $\langle v, val'\rangle$ *of v associated with a non-empty resp-entry,* $resp[\langle v, val'\rangle]$ *is a responsibility set of* $nogood(\langle v, val'\rangle)$*. Then* $CurSol \cup \langle u, val\rangle$ *is a nogood with a responsibility set* $RS \cup \{v\}$ *where RS is the union of responsibility sets of all the removed values of v.*

Proof Assume that the *resp*-entries associated with all the removed values of v contain $\emptyset$. This means that all the removed values of v are incompatible with *CurSol*. Considering that $\langle u, val\rangle$ conflicts with all the values of the *current* domain of v, we obtain that $CurSol \cup \{\langle u, val\rangle\}$ conflicts with all the values of the *original* domain of v. That is $CurSol \cup \{\langle u, val\rangle\}$ is a nogood with a responsibility set $\{v\}$.

On the other hand, if the domain of v contains values with non-empty *resp*-entries, let $\langle v, val'\rangle$ be such a value. Observe that $nogood(\langle v, val'\rangle) \subseteq CurSol \cup \{\langle u, val\rangle, \langle v, val'\rangle\}$. This means that the latter cannot be extended to a partial solution that assigns all the variables of $resp[\langle v, val'\rangle] \cup RS$. It follows that $CurSol \cup \{\langle u, val\rangle\}$ cannot be extended to a partial solution that assigns all the variables of $RS \cup \{v\}$ □

Proposition 5 *Consider a state that occurs during execution of MAC-RKP. Let* $\langle u, val\rangle$ *be a value removed by backtracking. Assume that every variable of* $ker[\langle u, val\rangle]$ *is neutral with respect to* $\langle u, val\rangle$*. Then CurSol is a nogood with a responsibility set computed by the mac_rkp_check_neutrality function regarding* $\langle u, val\rangle$*.*

Proof Let R be the responsibility set returned by *mac_rkp_check_neutrality* as the second component of the $FAIL$ message, when it detects that the pruning condition is satisfied regarding $\langle u, val\rangle$. Similarly to the proof of Proposition 3, we assume by contradiction that there is a partial solution P that contains *CurSol* and assigns all the variables of R. If P assigns all the variables of $ker[\langle u, val\rangle]$ with the values of their

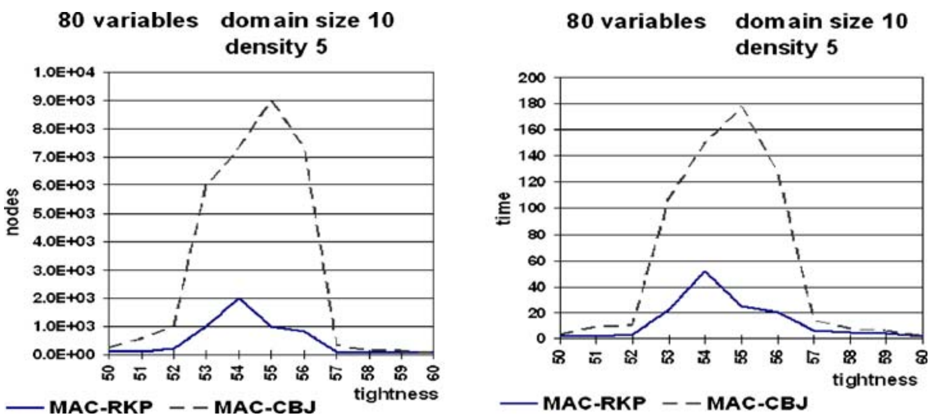

Fig. 5 Random CSPs: 5% density

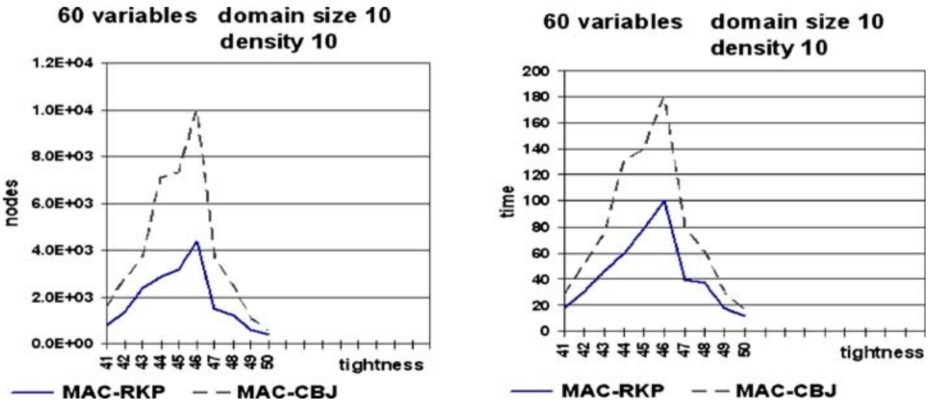

Fig. 6 Random CSPs: 10% density

current domains, let P' be the subset of P that contains *CurSol* and all the variables of *resp*$[\langle u, val\rangle]$. The same argumentation that was used in the proof of Proposition 3 shows that P' is inconsistent and, consequently, that P is inconsistent as well.

Now, assume that some variable $w \in ker[\langle u, val\rangle]$ is assigned with a value $\langle w, val'\rangle$ that does not belong to the current domain of w. Note that for FC-RKP such an assumption is incorrect because $\langle w, val'\rangle$ would be incompatible with *CurSol*. However, in the case of MAC-RKP, the assumption is valid since $\langle w, val'\rangle$ can simply be removed during the enforcement of AC.

Observe that $nogood(\langle w, val'\rangle) \subseteq CurSol \cup \{\langle w, val'\rangle\}$ and that all the variables of *resp*$[\langle w, val'\rangle]$ are assigned by P. Let P'' be a subset of P that contains $nogood(\langle w, val'\rangle)$ and assigns all the variables of *resp*$[\langle w, val'\rangle]$. By definition of a responsibility set, P'' is inconsistent, and so is P. □

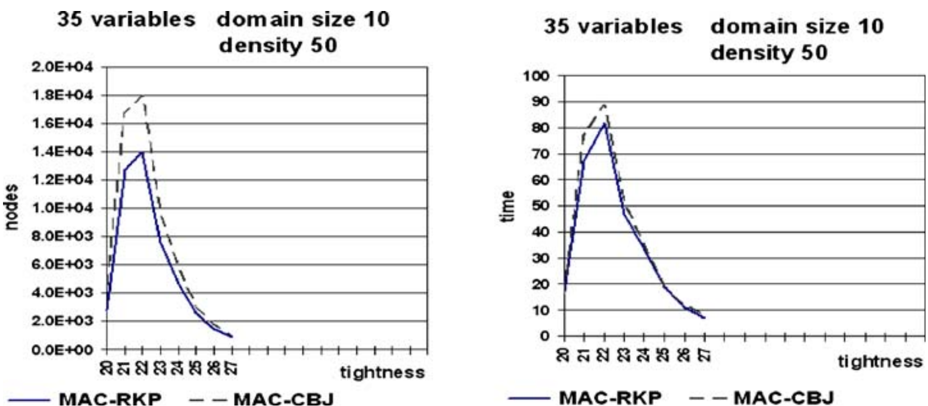

Fig. 7 Random CSPs: nodes visited for 50% density

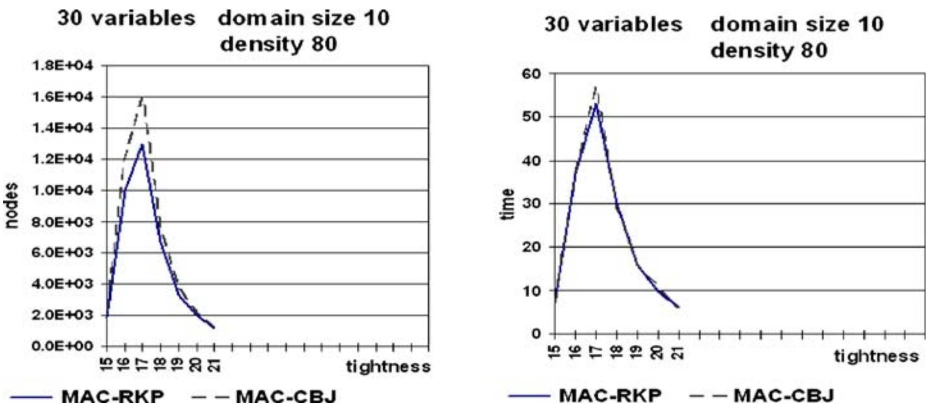

Fig. 8 Random CSPs: nodes visited for 80% density

7 Experimental Evaluation

To evaluate the proposed pruning technique, we compare MAC-RKP to MAC-CBJ [12].[2] The algorithms are applied to randomly generated CSPs and to random Graph k-Coloring problems. Two measures of performance are used: the number of nodes visited and the runtime. For the comparison, it is preferable to use runtime than the number of consistency checks because the latter does not take into account additional computational overhead. Both of the algorithms order variables by the Fail-First heuristic [8] which selects a variable with the smallest current domain size. The values within each variable are ordered by the min-conflict heuristic [5]. Every measure is obtained as an average of 50 runs.

Random CSPs are generated given their number of variables, domain size, density p_1 and tightness p_2 [13]. We examine four sets of instances by fixing the former three parameters and varying the tightness over the whole [0, 1] range, to get problems of all possible difficulties [13]. The resulting graphs are shown in Figs. 5–10.

The results for each set of parameters are represented by two graphs: one on the left side and the other on the right. The left graph compares the number of nodes visited, the right-hand side one compares the runtime. The solid-line graphs represent the behaviour of MAC-RKP and the dotted-line graphs represent the behaviour of MAC-CBJ.

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

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

[2]The results of comparing FC-RKP to FC-CBJ are similar. We omit them for the sake of brevity.

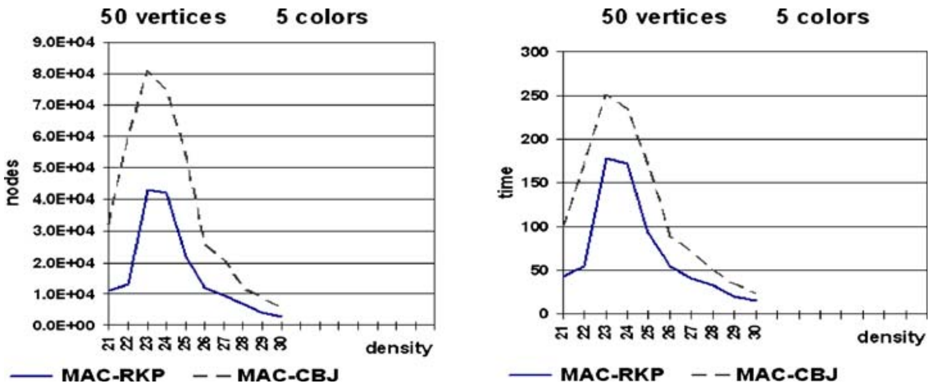

Fig. 9 Graph *k*-Coloring: hard problems of low density

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 CSPs with low density, every value is constrained with a relatively small number of variables, which, in turn, causes small kernels to be produced during the execution of MAC-RKP.

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

The proposed pruning methods and the resulting algorithms turn out to be especially efficient for Graph *k*-Coloring problems. We generate random Graph *k*-Coloring problems specifying the number of vertices, the number of colors and the density. The resulting CSP has the set of variables corresponding to the set of vertices, the domain of every variable corresponds to the set of colors. The pairs of

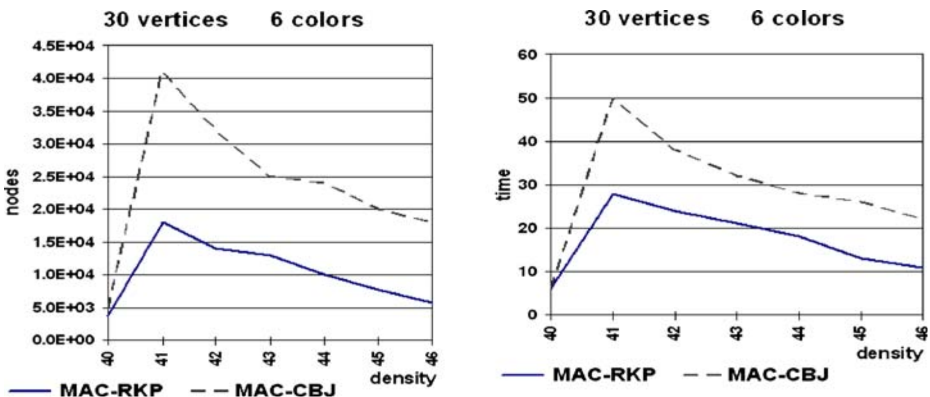

Fig. 10 Graph *k*-Coloring: hard problems of medium density

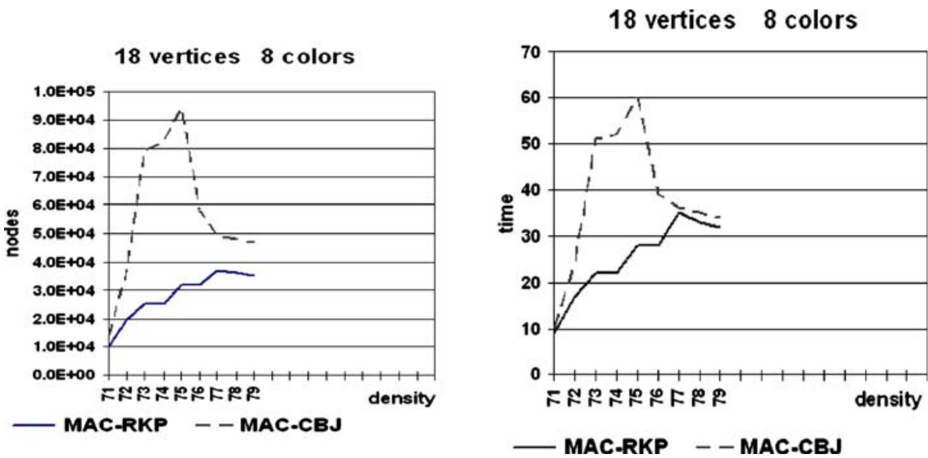

Fig. 11 Graph *k*-Coloring: hard problems of high density

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

We compare the algorithms on three sets of instances. For every set of instances, we fix the number of vertices and the number of colors and vary the density. The fixed parameters are selected as follows. In the first set of instances the phase transition region occurs for small densities. In the second set, phase transition occurs for medium densities and in the last set, the hardest instances have a high density. The results are presented in Figs. 9–11.

It is easy to see that the advantage of MAC-RKP over MAC-CBJ is very pronounced for the instances of Graph *k*-Coloring problems. MAC-RKP outperforms MAC-CBJ by a factor of 1.5 in runtime on the hard problems of low density (see Fig. 9). When the region of hard Graph *k*-Coloring instances is of higher density, (see Figs. 10 and 11) the improvement factor in runtime becomes greater than 2.

8 Conclusion

In this paper we presented an approach to pruning during CSP search. This approach uses new data structures called responsibility sets and kernels. 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 conflict sets and thus they connect two different areas of constraint reasoning: pruning "by analogy" [2–4], and intelligent backtracking [1, 7, 9, 11].

One can 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 CSPs with inequality constraints using domain-dependent features of such CSPs. Two facts 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 [17]. 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.

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 [10]. 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.

Acknowledgements Partially supported by the Lynn and William Frankel Center for Computer Science and by the Paul Ivanier Center for Robotics.

References

1. Bacchus, F. (2000). Extending forward checking. In *Principles and practice of constraint programming* (pp. 35–51).
2. Choueiry, B., & Noubir, G. (1998). On the computation of local interchangeability in discrete constraint satisfaction problems. In *Proceedings of AAAI* (pp. 326–333). Menlo Park, CA: AAAI.
3. Fahle, T., Schamberger, S., & Sellmann, M. (2001). Symmetry breaking. In *CP2001* (pp. 93–108). Berlin: Springer.
4. Focacci, F., & Milano, M. (2001). Global cut framework for removing symmetries. In *CP2001* (pp. 93–108). Berlin: Springer.
5. Frost, D., & Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95* (pp. 572–578). Montreal, Canada.
6. Gent, I., MacIntyre, E., Prosser, P., Smith, B., & Walsh, T. (1996). An empirical study of dynamic variable ordering heuristics. In *CP-96* (pp. 179–193).
7. Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research, 1*, 25–46.
8. Haralick, R. M., & Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence, 14*, 263–313.
9. Jussien, N., Debruyne, R., & Boizumault, P. (2000). Maintaining arc-consistency within dynamic backtracking. In *Principles and practice of constraint programming (CP 2000)* (pp. 249–261). Singapore, Springer
10. Kautz, H., & Selman, B. (2003). Ten challenges redux: Recent progress in propositional reasoning and search. In *CP2003* (pp. 1–18). Berlin: Springer.
11. Prosser, P. (1993) Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence, 9*, 268–299.
12. Prosser, P. (1995). MAC-CBJ: Maintaining Arc Consistency with Conflict-directed Backjumping. Technical Report, Research Report/95/177, Department of Computer Science, University of Strathclyde.
13. Prosser, P. (1996). An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence, 81*, 81–109.
14. Puget, J. (2005). Symmetry breaking revisited. *Constraints, 10*(1), 23–46.
15. Quimper, C.-G., Lopez-Ortiz, A., vanBeek, P., & Golynski, A. (2004). Improved algorithms for the global cardinality constraint. In *Principles and practice of constraint programming-CP2004, Toronto, Canada* (pp. 542–556). Berlin: Springer.
16. Razgon, I., & Meisels, A. (2003). Maintaining dominance consistency. In *Principles and practice of constraint programming-CP2003, Kinsale, Ireland* (pp. 945–950). Berlin: Springer.

17. Razgon, I., & Meisels, A. (2004). Pruning by equally constrained variables. In *Proceedings of CSCLP 2004* (pp. 26–40).
18. 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). Menlo Park, CA: American Association for Artificial Intelligence.
19. Sabin, D., & Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *PPCP'94* (pp. 10–20).
20. Wallace, R. (2005). Analysis of heuristic synergies. In *CSCLP 2005* (pp. 1–13).