M.R.C. van Dongen (Ed.)

Constraint Propagation and Implementation

First International Workshop
Toronto, Canada, September 2004
Proceedings

Held in conjunction with the
Tenth International Conference on
Principles and Practice of
Constraint Programming (CP 2004)

Preface

Constraint Propagation is an essential part of many constraint programming systems. Sitting at the heart of a constraint solver, it consumes a significant portion of the time that is required for problem solving.

The "First International Conference on Constraint Propagation and Implementation (CPAI'2004)" was convened to study the design and analysis of new propagation algorithms as well as practical issues in and the evaluation of implementing existing and new constraint propagation algorithms in settings ranging from special purpose solvers to programming language systems.

These proceedings contain the thirteen contributed papers presented as talks or posters at the workshop.

We wish to thank the authors for submitting their work, the invited speaker, Jean-Charles Régin, the CP 2004 Workshop/Tutorial Chair, Barry O'Sullivan, the members of the CPAI'2004 Programme Committee, and the additional reviewers.

August 2004

Marc van Dongen
Yuanlin Zhang
Rick Wallace

Organising Committee

Programme Committee

Members of Programme Committee

Organiser	Marc van Dongen (4C, Ireland)
Organiser	Yuanlin Zhang (4C, Ireland)
Organiser	Rick Wallace (4C, Ireland)
	Christian Bessière (LIRMM-CNRS, France)
	Warwick Harvey (IC-Parc, Imperial College London, UK)
	Willem Jan van Hoeve (CWI, Holland)
	Christophe Lecoutre (Université d'Artois, France)
	Jean-Charles Régin (ILOG, France)
	Peter van Beek (University of Waterloo, Canada)
	Pascal Van Hentenryck (Brown University, USA)

Additional Reviewers

Diarmuid Grimes (4C, Ireland)
Mark Hennessy (4C, Ireland)
Chavalit Likitvivatanavong (4C, Ireland)
Deepak Mehta (4C, Ireland)

Table of Contents

A Constraint Algebra

Fahiem Bacchus[1]* and Toby Walsh[2]**

[1] Department of Computer Science, University of Toronto, Canada. fbacchus@cs.toronto.ca
[2] Cork Constraint Computation Center, University College Cork, Ireland. tw@4c.ucc.ie

Abstract. Many constraint toolkits provide logical connectives like disjunction, negation and implication that permit complex constraint expressions to be built from more primitive constraints. However, the propagation of complex constraints is typically delayed and so give little pruning. In this paper, we present a simple and light weight way to propagate such constraint expressions. We prove that computing the maximal set of inconsistent assignments for a constraint expression is intractable in general. We therefore provide a polynomial time function which computes a tractable subset compositionally. We characterise precisely when this function computes maximal inconsistent sets. In such cases, our function enforces generalized arc-consistency on the constraint expression. We then lift the reasoning from inconsistent assignments to inconsistent bounds of integer, set or multiset variables. Finally, as our approach requires being able to compute valid as well as inconsistent assignments and bounds for primitive constraints, we demonstrated that valid assignments and bounds can easily be computed for many primitive constraints.

1 Introduction

To facilitate the modeling of problems as constraint programs, constraint toolkits provide a wide range of primitive constraints along with propagators for these constraints. However, only limited mechanisms for combining these primitive constraints are available, and typically such combinations are not propagated very effectively. For example, whilst many toolkits permit disjunctions of constraints, the propagation of such disjunctions is generally delayed until all but one of the disjunctions are disentailed (and the remaining disjunct must hold).

In this paper, we discuss a simple algebra for combining primitive constraints. We build complex propositional constraint expressions from primitive constraints using the logical connectives conjunction, disjunction and negation. Expressions involving implication, equivalence, exclusive-or, etc., can therefore also be represented. We show how a simple extension to constraint propagators will permit such complex constraint expressions to be propagated in a compositional way. Such an algebra can therefore be incorporated into any current constraint toolkit by simply extending the propagators for the primitive constraints. We can therefore provide the user with a rich language for specifying their problem, whilst preserving some of the propagating power of the primitive constraints.

* Supported by Natural Science and Engineering Research Council Canada.
** Supported by Science Foundation Ireland.

The paper is structured as follows. We first present some background. We then define a simple constraint algebra and discuss the complexity of making an arbitrary constraint expression in this algebra generalized arc consistent (GAC). A polytime algorithm for computing inconsistent assignments for a constraint expression is provided. This function is compositional, and computes inconsistent assignments from the inconsistent and valid assignments of the primitive constraints within the constraint expression. We then characterise when this function computes maximal inconsistent sets. In such cases, pruning the values computed by this function will ensure that the constraint expression is GAC. Following this, we show how bounds consistency can be achieved on constraint expressions involving integer, set or multiset variables. To apply our approach, we need to extend the propagators of primitive constraints so that they can compute valid as well as inconsistent assignments and bounds. A wide range of examples is given to demonstrate that this is typically not difficult to achieve. We end with a discussion of related work and some conclusions.

2 Inconsistent and Valid assignments

A *constraint satisfaction problem* is a set of variables, each with a finite domain of values, and a set of constraints that specify allowed values for subsets of variables. Each constraint consists of a relation of allowed values and a scope of variables to which the constraint is applied. For convenience, we represent the domains $\mathcal{D}$of the variables in a problem by a set of possible assignments. For example, if we have just two variables X and Y with 0/1 domains, then $\mathcal{D} = \{X = 0, X = 1, Y = 0, Y = 1\}$. We let $domain(X)$ be the set of values in the domain of the variable X. That is, $domain(X) = \{a \mid X = a \in \mathcal{D}\}$. An **assignment set** τ is a set of assignments to variables such that every variable X in $scope(\tau)$ (the set of variables appearing in τ) is assigned only one value from $domain(X)$. We also use $scope(C)$ for the variables in the constraint C. Given a constraint C and an assignment set τ with $scope(C) \subseteq scope(\tau)$, we write $C(\tau)$ iff the assignments in τ satisfies C. That is $C(\tau)$ iff there exists $X_1 = a_1, \ldots X_k = a_k \in \tau$ with $scope(C) = \{X_1, \ldots, X_k\}$ and $X_1 = a_1, \ldots X_k = a_k$ satisfies C. We write $\neg C(\tau)$ otherwise.

An assignment is (generalized arc) **inconsistent** for a constraint iff all assignment sets containing it fail to satisfy the constraint. That is, $X = a$ is inconsistent for C iff $\forall \tau. \big(scope(C) \subseteq scope(\tau) \wedge X = a \in \tau\big) \rightarrow \neg C(\tau)$. A constraint C has an unique maximal set of inconsistent assignments $MaxInc(C)$. For example, given the constraint $X < Y$ with $X = \{0, 1, 2\}$ and $Y = \{1, 2\}$, then $\{X = 2\}$ is the maximal set of inconsistent assignments. No possible extension of $X = 2$ satisfies the constraint $X < Y$, but all other assignments can be extended to satisfy the constraint. Assignments that are **consistent** have at least one witness falsifying the above condition; i.e., $X = a$ is consistent iff there is an assignment set τ (called a **support**) with $scope(\tau) = scope(C) \wedge X = a \in \tau \wedge C(\tau)$. A constraint C is **GAC** (Generalized Arc Consistent) iff every value of every variable in $scope(C)$ has at least one support.

If $X = a$ is inconsistent, we can prune a from the domain of X. It is well known that a constraint C can be made GAC by simply pruning all values in $MaxInc(C)$ from the domains of their respective variables. This process might reduce the domains of all variables to the empty set, achieving GAC in a trivial way.

Valid assignments are the dual of inconsistent assignments, and are essential in the algebra we develop. An assignment is **valid** wrt a constraint iff all assignment sets containing that assignment satisfies the constraint. That is, $X = a$ is valid for C iff $\forall\tau.\big(scope(C) \subseteq scope(\tau) \wedge X = a \in \tau\big) \rightarrow C(\tau)$. As with inconsistent values every constraint C has a unique maximal set of valid assignments, $MaxValid(C)$. For example, given the constraint $X < Y$ with $X = \{0, 1, 2\}$ and $Y = \{1, 2\}$, the maximal set of valid assignments is $\{X = 0\}$. All possible extensions of $X = 0$ satisfy the constraint $X < Y$, but no other assignment always satisfies the constraint. All of the concepts presented for inconsistent assignments have dual versions for valid assignments. For example, the dual of consistent assignments is the notion of non-valid assignments. An assignment $X = a$ is **non-valid** if there is at least one assignment set τ with $scope(\tau) = scope(C) \wedge X = a \in \tau \wedge \neg C(\tau)$. Another example is that the dual of a support is a non-support. An assignment set τ is a **non-support** for an assignment $X = a$ in a constraint C iff $scope(\tau) = scope(C) \wedge X = a \in \tau \wedge \neg C(\tau)$. A non-support witnesses the non-validity of $X = a$.

3 A constraint algebra

To build complex constraints, we combine primitive constraints using the propositional connectives negation, disjunction and conjunction. A **constraint expression** is either a primitive constraint C or any well-founded Boolean expression of the form: $true$, $false$, $not(\mathcal{C}_1)$, $or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ or $and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$, where each $\mathcal{C}_i$ is itself a constraint expression. $true$ is the primitive constraint which is always valid, whilst $false$ is always inconsistent. We also allow the expressions $implies(\mathcal{C}_1, \mathcal{C}_2)$, $iff(\mathcal{C}_1, \mathcal{C}_2)$, $xor(\mathcal{C}_1, \mathcal{C}_2)$ and $ifthen(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$, but regard these additional connectives simply as abbreviations. More formally:

$$
\begin{aligned}
implies(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow or(not(\mathcal{C}_1), \mathcal{C}_2) \\
iff(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow and(or(not(\mathcal{C}_1), \mathcal{C}_2), or(not(\mathcal{C}_2), \mathcal{C}_1) \\
xor(\mathcal{C}_1, \mathcal{C}_2) &\leftrightarrow and(or(\mathcal{C}_1, \mathcal{C}_2), or(not(\mathcal{C}_1), not(\mathcal{C}_2)) \\
ifthen(\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3) &\leftrightarrow and(or(not(\mathcal{C}_1), \mathcal{C}_2), or(\mathcal{C}_1, \mathcal{C}_3))
\end{aligned}
$$

Each constraint expression $\mathcal{C}$ represents a new constraint whose scope is equal to the union of the scopes of the primitive constraints in $\mathcal{C}$. An assignment set τ satisfies $\mathcal{C}$ iff $scope(\mathcal{C}) \subseteq scope(\tau)$ and the Boolean expression representing $\mathcal{C}$ evaluates to true given the truth values of the component primitive constraints under τ. For example, an absolute value constraint $X = abs(Y)$ can be written as the constraint expression $ifthen(Y \geq 0, X = Y, X = -Y)$. Similarly, a max constraint $X = max(Y, Z)$ can be written as the constraint expression $and(X \geq Y, X \geq Z, or(X = Y, X = Z))$. Since a constraint expression is itself a constraint, associated with every constraint expression $\mathcal{C}$ is a maximal set of inconsistent assignments, $MaxInc(\mathcal{C})$, and a maximal set of valid assignments $MaxValid(\mathcal{C})$. We can make the constraint expression $\mathcal{C}$ GAC by pruning all assignments in $MaxInc(\mathcal{C})$. It is also useful to observe that a simple consequence of the duality between valid assignments and inconsistent assignments is that $MaxInc(\mathcal{C}) = MaxValid(not(\mathcal{C}))$.

Constraint propagation, the detection and deletion of inconsistent values, is one of the central aspects of constraint programming. How then do we propagate constraint expressions? Clearly we do not want to design new propagators for every new constraint expression: this would defeat the usefulness of the algebra. First we examine the complexity of computing all valid and inconsistent assignments for an arbitrary constraint expression.

3.1 Computational complexity.

The following theorem demonstrates that it is in general intractable to compute $MaxInc(\mathcal{C})$ or $MaxValid(\mathcal{C})$ for an arbitrary constraint expression.

Theorem 1. *Determining if an assignment is in $MaxInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k))$ is coNP-Complete as is determining if an assignment is in $MaxValid(or(\mathcal{C}_1, \ldots, \mathcal{C}_k))$. The hardness does not change when the input of the problem includes $MaxInc(\mathcal{C}_i)$ and $MaxValid(\mathcal{C}_i)$ for all $i \in \{1, \ldots, k\}$.*

Proof: We show that the complementary problem of deciding if $X = a \notin MaxInc(\mathcal{C})$ for $\mathcal{C} = and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ is NP-Complete. It is clearly in NP since an assignment set τ with $\mathcal{C}(\tau)$ and $X = a \in \tau$ is a poly-time checkable witness. A reduction from 3SAT allows us to show completeness. We map the Boolean variables x_i in the SAT problem φ to 0/1 variables X_i in the CSP so that $x_i =$ *true* iff $X_i = 1$. We also include a new 0/1 variable, X, in the CSP. We construct $and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ where $\mathcal{C}_i$ represents the i-th clause of the 3SAT problem augmented with the new variable X. For example, if the i-th clause is $(x_3, \neg x_5, x_7)$, then $\mathcal{C}_i$ will be $or(C(X_3), not(C(X_5)), C(X_7), C(X))$ where each $C(X_i)$ is a unary constraint that is satisfied iff $X_i = 1$. Suppose $X = 0$ is not in the maximal set of inconsistent assignments. Then there are assignments to all the other variables that satisfies each of the constituent constraint expressions $\mathcal{C}_i$. Thus, there is an assignment set that satisfies each clause. As the proof reverses immediately, $X = 0$ is not in the maximal set of inconsistent assignments iff φ is satisfiable. Determining if an assignment is not in the maximal set of inconsistent assignments is therefore NP-complete. By duality $MaxValid(\mathcal{C}) = MaxInc(not(\mathcal{C}))$. Hence, determining $MaxValid$ for the above expression is polynomially reducible to a co-NP-complete problem, and it also is co-NP-complete.

In either case, for each of the clause constraints $\mathcal{C}_i$, $MaxValid(\mathcal{C}_i)$ and $MaxInc(\mathcal{C}_i)$ are both empty: any assignment to any of the variables in $\mathcal{C}_i$ has a support and a non-support. Supplying this information as input cannot make the computation easier, as these sets are always empty for any 3SAT problem. ∎

Whilst deciding if an assignment is inconsistent is coNP-complete, deciding if a set of assignments is *the maximal* inconsistent set for an arbitrary constraint expression is D^P-complete in general. The complexity class D^P contains problems which are the conjunction of a problem in NP and one in coNP [9]. It is also known as the second level of the Boolean hierarchy, BH_2(NP).

Theorem 2. *Determining if a set is the maximal set of inconsistent assignments for a conjunction $and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ is D^P-complete as is determining if a set is the maximal set of valid assignments for a disjunction $or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$.*

Proof: We show that deciding that a set is the maximal set of inconsistent assignments answers positively the NP-complete problem of determining if a 3-cnf formula φ_1 is SAT, and the coNP-complete problem of determining if a 3-cnf formula φ_2 is UNSAT. Using the same construction as Theorem 1, we represent the clauses in φ_1 with the constraint expressions $\mathcal{C}_i$ containing the new variable X, and the clauses in φ_2 with the constraint expressions $\mathcal{D}_i$ containing the new variable Y. Then we construct the constraint expression $and(\mathcal{C}_1, \ldots, \mathcal{C}_j, \mathcal{D}_1, \ldots, \mathcal{D}_k)$. $\{Y = 0\}$ is the maximal set of inconsistent assignments iff φ_1 is SAT and φ_2 is UNSAT. A dual argument shows that recognizing whether a set of assignments is the maximal set of valid assignments is also D^P-complete. ∎

Computing valid or inconsistent assignments for complex constraint expressions can thus be computationally harder than computing such assignments for the constituent parts. However, it also can be easier. For instance, there exist classes of constraints for which recognizing if an assignment is inconsistent is coNP-complete, but recognizing if an assignment is inconsistent for their conjunction is polynomial. Consider $\sum_i X_i = k$ and $\sum_i X_i = k + 1$. Identifying consistent assignments for such constraints is NP-complete using a simple reduction from subset sum. However, their conjunction is trivially inconsistent.

The complexity of determining $MaxInc(\mathcal{C})$ for an arbitrary constraint expression is also closely related to the complexity of query evaluation in relational databases [10, 11]. In particular, if $scope(\mathcal{C}) = \{X, Y_1, \ldots, Y_k\}$, determining all assignments to X that are not in $MaxInc(\mathcal{C})$ is equivalent to evaluating the first-order query $\exists Y_1, \ldots, Y_k \mathcal{C}(X)$ in a database where the relations are the primitive constraints in $\mathcal{C}$. Thus $MaxInc(\mathcal{C})$ can be determined with $k + 1$ such queries. This particular type of query, however has not been studied in the database literature, and hardness results proved for more general classes of queries do not entail our results.[3]

4 Computing inconsistent assignments

Since computing the maximal set of inconsistent assignments for a complex constraint expression is intractable in general, we propose a simple and light weight method for computing a subset of the maximal sets in polynomial time. The method is compositional as it computes the inconsistent and valid assignments of a constraint expression in terms of the inconsistent and valid assignments of its parts. For a constraint expression $\mathcal{C}$ and variable domains $\mathcal{D}$, the functions $Inc(\mathcal{C}, \mathcal{D})$ and $Valid(\mathcal{C}, \mathcal{D})$ return subsets of $MaxInc(\mathcal{C})$ and $MaxValid(\mathcal{C})$ respectively. These functions recursively apply the rules in Table 1, until they reach the primitive constraints. We then assume that each primitive constraint has a poly-time algorithm to compute inconsistent and valid assignments. In section 7 we show that this is a reasonable assumption by demonstrating such algorithms for a number of different primitive constraints.

We prove that these functions are correct. That is, they compute assignments which are indeed inconsistent and valid.

[3] There are also a number of results on tractable queries that do apply to the problem of computing $MaxInc$ and $MaxValid$. We are more interested here in approximating $MaxInc$ and $MaxValid$ for general (intractable) constraint expressions, so we refer the reader to [11] for more details on these tractable cases.

$Inc(not(\mathcal{C}_1), \mathcal{D})$	$= Valid(\mathcal{C}_1, \mathcal{D})$
$Valid(not(\mathcal{C}_1), \mathcal{D})$	$= Inc(\mathcal{C}_1, \mathcal{D})$
$Inc(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$	$= \bigcap_{1 \leq i \leq k} Inc(\mathcal{C}_i, \mathcal{D})$
$Valid(and(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$	$= \bigcap_{1 \leq i \leq k} Valid(\mathcal{C}_i, \mathcal{D})$
$Inc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$	$= IterativeInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$
$Valid(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$	$= IterativeValid(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$

$IterativeInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$

```
inc := ⋃_{1≤i≤k} Inc(C_i, D)
repeat
    D      := D − inc
    nxtinc := ⋃_{1≤i≤k} Inc(C_i, D)
    inc    := inc ∪ nxtinc
until(nxtinc = ∅)
    return(inc)
```

$IterativeValid(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$

```
valid := ⋃_{1≤i≤k} Valid(C_i, D)
repeat
    D       := D − valid
    nxtvalid := ⋃_{1≤i≤k} Valid(C_i, D)
    valid   := valid ∪ nxtvalid
until(nxtvalid = ∅)
    return(valid)
```

Table 1. Functions for computing valid and inconsistent assignments of a constraint expression. In addition, $Inc(true, \mathcal{D}) = Valid(false, \mathcal{D}) = \emptyset$, and $Valid(true, \mathcal{D}) = Inc(false, \mathcal{D}) = \mathcal{D}$.

Theorem 3. *For any primitive constraint C_i, if for all subsets $\mathcal{D}'$ of $\mathcal{D}$, $Inc(C_i, \mathcal{D}')$ contains only inconsistent assignments and $Valid(C_i, \mathcal{D}')$ contains only valid assignments, then $Inc(\mathcal{C}, \mathcal{D})$ and $Valid(\mathcal{C}, \mathcal{D})$ contain only inconsistent and valid assignments respectively for any constraint expression built from these primitive parts.*

Proof: By induction on the structure of the constraint expression. The base case holds by assumption. The step case uses case analysis. For a constraint expression $not(\mathcal{C}_1, \mathcal{D})$, we have that $Inc(not(\mathcal{C}_1), \mathcal{D}) = Valid(\mathcal{C}_1, \mathcal{D})$. By induction, the assignments in $Valid(\mathcal{C}_1, \mathcal{D})$ are valid. Hence all of these assignments are inconsistent for $not(\mathcal{C}_1, \mathcal{D})$. A dual argument shows that the assignments in $Valid(not(\mathcal{C}_1), \mathcal{D})$ are valid. For a constraint expression $\mathcal{C} = and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$, we have $Valid(\mathcal{C}, \mathcal{D}) = \bigcap_{1 \leq i \leq k} Valid(\mathcal{C}_i, \mathcal{D})$. Suppose $X = a \in Valid(\mathcal{C}, \mathcal{D})$. Then for all i, $X = a \in Valid(\mathcal{C}_i, \mathcal{D})$. By the induction hypothesis, the assignments in each $Valid(\mathcal{C}_i)$ are valid. Consider any assignment set τ such that $X = a \in \tau$ and $scope(\mathcal{C}) \subseteq scope(\tau)$. Since $X = a$ is valid for each $\mathcal{C}_i$, τ must satisfy all $\mathcal{C}_i$ and thus must satisfy the conjunction. Hence $X = a$ is also valid for $\mathcal{C}$. For $\mathcal{C} = Inc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$ a similar argument shows that the assignments in $\bigcup_{1 \leq i \leq k} Inc(\mathcal{C}_i, \mathcal{D})$ are inconsistent. Deleting these assignments from $\mathcal{D}$ cannot cause any consistent assignment to lose its support, hence $Inc(\mathcal{C}_i, \mathcal{D}')$ on this reduced domain $domain'$ must still return inconsistent assignments. $IterativeInc$ then re-computes $\bigcup_{1 \leq i \leq k} Inc(\mathcal{C}_i, \mathcal{D})$ until we reach a fixed point.

Similar arguments also hold for constraint expressions of the form $or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$. In this case, deleting valid assignments from $\mathcal{D}$ cannot cause any non-valid assignment to lose its non-support and $Valid(\mathcal{C}_i)$ on the reduced domain must still return valid assignments. ∎

The algorithm can be optimized by the simple caching scheme in which we remember the previously computed value $Inc(\mathcal{C}_i, \mathcal{D})$ for each subexpression $\mathcal{C}_i$. If in a subsequent call $Inc(\mathcal{C}_i, \mathcal{D}')$, $\mathcal{D}'$ is identical to $\mathcal{D}$ when restricted to the variables in

$scope(\mathcal{C}_i)$, then we can reuse the previously computed result for $Inc(\mathcal{C}_i, \mathcal{D})$. A similar optimization works for $Valid(\mathcal{C}_i, \mathcal{D})$. In addition, if we compute and prune inconsistent values incrementally, we can stop as soon as any variable has a domain wipeout.

4.1 Termination

The $IterativeInc$ and $IterativeValid$ functions may take a number of iterations to reach their fixed point. In practice, this fixed point is likely to be reached after only a few iterations. It is possible to show that only a polynomial number of iterations is ever required.

Theorem 4. *$IterativeInc$ and $IterativeValid$ take $O(nd)$ iterations to reach their fixed points for constraint expressions with n variables and domains of size d. There exist constraint expressions on which they take $\Theta(nd)$ iterations to reach its fixed point.*

Proof: As each iteration removes at least one value, we must reach the fixed point in at most nd steps. We can give a simple example in which this bound is reached. Consider $and(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ where $\mathcal{C}_i$ is $X_i = X_{i+1}$ for $i < n$ and $X_1 - X_n = 1$ for $i = n$. Suppose $domain(X_i) = \{1, \ldots, d\}$ for every i. Then in the first iteration, $IterativeInc$ returns $\{X_1 = 1\}$ as this value is not supported in $X_1 - X_n = 1$. After this is pruned from $\mathcal{D}$, a second iteration returns $\{X_2 = 1\}$ as this value is now not supported in $X_1 = X_2$. And so on up to the nth iteration which returns $\{X_n = 1\}$. After this is pruned, the $n + 1$th iteration returns $\{X_1 = 2\}$ as this value is now not supported in $X_1 - X_n = 1$. Hence, there are nd iterations before all the values of all the variables are removed. Note that even if we stop when the first variable has a domain wipeout, it will still take $(n - 1)d + 1$ iterations before the first variable has a domain wipeout. ∎

4.2 Entailment and disentailed

A constraint expression is entailed iff it holds for all possible assignments. A constraint is disentailed iff it does not hold for any possible assignment. If $Valid(\mathcal{C}, \mathcal{D})$ equals the domains of all of the variables in the scope of $\mathcal{C}$, then $\mathcal{C}$ is entailed. In such a situation, we modify the computation of $Valid$ so that $Valid(\mathcal{C}, \mathcal{D}) = \mathcal{D}$. Similarly, if $Inc(\mathcal{C}, \mathcal{D})$ equals the domains of all of the variables in the scope of $\mathcal{C}$ then $\mathcal{C}$ is disentailed. We now modify the computation of Inc so that $Inc(\mathcal{C}, \mathcal{D}) = \mathcal{D}$.

To show the benefit of detecting (dis)entailment in this way, consider the constraint expression, $implies(even(X), odd(Y))$ and the domains $\mathcal{D} = \{X = 0, X = 2, Y = 1, Y = 2\}$. $Inc(implies(even(X), odd(Y)), \mathcal{D}) = Valid(even(X), \mathcal{D}) \cap Inc(odd(Y), \mathcal{D})$. Using the unmodified versions, $Valid(even(X), \mathcal{D})$ will return just valid values for X, whilst $Inc(odd(Y)), \mathcal{D})$ returns just inconsistent values for Y. Inc would then compute the empty set of inconsistent assignments for $implies(even(X), odd(Y))$. Note, however, that $domain(X)$ only contains even numbers. Hence $even(X)$ is entailed. Therefore the modified $Valid(even(X), \mathcal{D})$ can return $\mathcal{D}$, in which case $Inc(implies(even(X), odd(Y), \mathcal{D}) = \mathcal{D} \cap Inc(odd(Y), \mathcal{D}) = \{Y = 2\}$. This is the maximal set of inconsistent assignments, as required.

4.3 Maximality

These functions do not always compute maximal sets, even if maximal sets are computed for the primitive constraints from which they are composed. This is not surprising given that computing maximal sets is intractable in general. The following result precisely characterizes when Inc returns the maximal inconsistent set of assignments. In other words, the following result identifies exactly when pruning the values returned by Inc ensures that a constraint expression is GAC.

We start with a number of definitions. A *hypergraph* $\mathcal{H} = (\mathcal{H}, \mathcal{E}^{\mathcal{H}})$ is a set of vertices H and hyperedges E^H each of which is a subset of H. An **acyclic tree decomposition** [11] of a hypergraph $\mathcal{H}$ is a tree T satisfying two properties: (1) there is a one-to-one correspondence between the hyperedges of $\mathcal{H}$ and the nodes of T; the hyperedge corresponding to a tree node t is called t's label ($label(t)$); (2) for every vertex $v \in \mathcal{H}$ the set of nodes t of T such that $v \in label(t)$ form a subtree of T. The hypergraph of a conjunctive or disjunctive constraint expression, $\mathcal{C} = and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ or $\mathcal{C} = or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$, has the variables in $scope(\mathcal{C})$ as vertices and the sets of variables $scope(\mathcal{C}_i)$, $i = 1, \ldots, k$ as hyperedges. We will relax this definition to take account of (dis)entailment. If $\mathcal{C} = and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ then we ignore any subexpression that is entailed when constructing the hypergraph. Similarly, if $\mathcal{C} = or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ then we ignore any subexpression that is disentailed. Under this relaxation we define a conjunctive or disjunctive constraint expression to be **acyclic** if its corresponding hypergraph has an acyclic tree decomposition. For example, a conjunction in which the primitive constraints are in a chain, and each has only one variable in common with the previous and next constraint is acyclic. Acyclic constraint expressions are, however, more general than chains. We use this notion of acyclicity to characterise when Inc computes the maximal set of inconsistent assignments.

Theorem 5. *For any constraint expression $\mathcal{C}$ and any variable domains $\mathcal{D}$, $Inc(\mathcal{C}, \mathcal{D}) = MaxInc(\mathcal{C}, \mathcal{D})$ if:*

1. *$\mathcal{C}$ is a primitive constraint and $Inc(\mathcal{C}, \mathcal{D}) = MaxInc(\mathcal{C}, \mathcal{D})$;*
2. *$\mathcal{C} = not(\mathcal{C}_1)$ and $Valid(\mathcal{C}_1, \mathcal{D}) = MaxValid(\mathcal{C}_1, \mathcal{D})$;*
3. *$\mathcal{C} = or(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ and $Inc(\mathcal{C}_i, \mathcal{D}) = MaxInc(\mathcal{C}_i, \mathcal{D})$ for $i \in (1, \ldots, k)$;*
4. *$\mathcal{C} = and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ and (a) $Inc(\mathcal{C}_i, \mathcal{D}) = MaxInc(\mathcal{C}_i, \mathcal{D})$ for $i \in (1, \ldots, k)$; (b) $\mathcal{C}$ is acyclic; and (c) $|scope(\mathcal{C}_i) \cap scope(\mathcal{C}_j)| \leq 1$ for $i, j \in (1, \ldots, k)$;*
5. *$Inc(C, \mathcal{D}) = \mathcal{D}$.*

Proof: 1. Immediate. 2. Suppose $Valid(\mathcal{C}_1, \mathcal{D})$ is maximal. Then for any $X = a \notin Valid(\mathcal{C}_1, \mathcal{D})$, there exists τ with $X = a \in \tau$ and $\neg\mathcal{C}_1(\tau)$. Hence $X = a$ cannot be in $Inc(not(\mathcal{C}_1), \mathcal{D})$ as τ is one assignment that prevents it being inconsistent. Hence $Inc(not(\mathcal{C}_1), \mathcal{D})$ is maximal.

3. Suppose $Inc(\mathcal{C}_i, \mathcal{D})$ are maximal. Consider $X = a \notin Inc(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$. Then $X = a \notin \bigcap_{1 \leq i \leq k} Inc(\mathcal{C}_i, \mathcal{D})$. That is, $X = a \notin Inc(\mathcal{C}_j, \mathcal{D})$ for some $j \in (1, \ldots, k)$. As $Inc(\mathcal{C}_j, \mathcal{D})$ is maximal, there exists τ with $X = a \in \tau$ and $\mathcal{C}_j(\tau)$. Thus $X = a$ cannot be in $Inc(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$ as τ is one assignment that prevents it being inconsistent. Hence $Inc(or(\mathcal{C}_1, \ldots, \mathcal{C}_k), \mathcal{D})$ is maximal.

4. Suppose $X \in scope(\mathcal{C})$ with $X = a \in \mathcal{D}$, but $X = a \notin Inc(\mathcal{C}, \mathcal{D})$. We must show that $X = a \notin MaxInc(\mathcal{C}, \mathcal{D})$. Let $\mathcal{D}^c = \mathcal{D} - Inc(\mathcal{C}, \mathcal{D})$, i.e., the consistent assignments remaining in the variable domains. From Table 1 we observe that $Inc(\mathcal{C}_i, \mathcal{D}^c) = \emptyset$ and by condition (a) $MaxInc(\mathcal{C}_i, \mathcal{D}^c) = \emptyset$ for all $i \in (1, \ldots, k)$. Consider the acyclic tree decomposition associated with $\mathcal{C}$. Orient this tree so that the root is labeled with $scope(\mathcal{C}_i)$ for some $\mathcal{C}_i$ with $X \in scope(\mathcal{C}_i)$. Note that by property (2) of an acyclic tree decomposition and condition (c), each of the subtrees below $\mathcal{C}_i$ can have at most one variable in common with the other subtrees. Furthermore if two subtrees do have a variable in common that variable must be in the $scope(\mathcal{C}_i)$.

Since $X = a \in \mathcal{D}^c$ it must have some support τ on $\mathcal{C}_i$ such that $\tau \in \mathcal{D}^c$. Now we extend this support downwards in the tree decomposition to the children of $\mathcal{C}_i$: $\mathcal{C}_j^1, \ldots, \mathcal{C}_j^\ell$. Each such child $\mathcal{C}_j$ shares only one variable with $\mathcal{C}_i$, say Y, and Y must be assigned some value in τ, say $Y = b$. Since $Y = b \in \mathcal{D}^c$ it must have a support τ_j in $\mathcal{C}_j$ such that $\tau_j \in \mathcal{D}^c$. Thus we can extend τ to a support for $and(\mathcal{C}_i, \mathcal{C}_j)$ for each child of $\mathcal{C}_i$. Furthermore the supports τ_j for the individual children of $\mathcal{C}_i$ cannot be in conflict: $\mathcal{C}_j$ and $\mathcal{C}_{j'}$ can only share a variable already assigned by τ, hence τ_j and $\tau_{j'}$ must agree with τ and with each other on the value assigned to this variable. Thus we can extend τ to a support for all of $\mathcal{C}_i$'s children. Furthermore, by the same argument each support τ_j for the child $\mathcal{C}_j$ can be extended to a support for all of the conjuncts in the subtree below $\mathcal{C}_j$. Hence, τ can be extended to a support for all of $\mathcal{C}$, and since $X = a \in \tau$, $X = a \notin MaxInc(\mathcal{C}, \mathcal{D})$.

Note that if $and(\mathcal{C}_1, \ldots, \mathcal{C}_k)$ contains any entailed conjuncts, these can be eliminated without changing the maximal set of inconsistent assignments. We can then apply the argument above to the remaining acyclic part of the conjunction.

5. Immediate since $MaxInc(\mathcal{C}, \mathcal{D}) \subseteq \mathcal{D}$. ∎

In fact, we can show that these 5 cases are the only ones in which Inc is always guaranteed to be maximal. This reverse direction needs a little care as Inc may compute $MaxInc$ by chance. However, these 5 cases are the only ones in which, irrespective of the constraint subexpressions, Inc is guaranteed to compute $MaxInc$. A dual result holds, and characterizes precisely when $Valid$ computes $MaxValid$.

Another way of viewing Inc is that when we run it on a conjunction of primitive binary constraints, $\mathcal{C} = and(C_1, \ldots, C_k)$, we enforce arc consistency. Furthermore, if $\mathcal{C}$ is acyclic then it has tree width 1. This is precisely the condition required for arc consistency to achieve GAC on the conjunction. For non-binary constraints the tree width of an acyclic expression is equal to the maximum scope of one of its conjunction (minus 1), and thus can be much larger than 1. Hence, we require the extra condition bounding the size of scope intersections to ensure GAC.

5 Bounds consistency

Set and multiset variables, and constraints upon them, are useful in many situations [4, 14]. We will therefore show how to deal with such variables. The central notion needed is bound consistency. We define this simultaneously for finite domain, set and multiset variables. We can therefore reason about constraints which involve finite domain, set and multiset variables. For example, we might wish to reason about a constraint like $or(|S| < N, |S| > M)$ which contains both a set variable, S and the finite domain

integer variables, M and N. In what follows, we assume that all finite domain variables are integers, but any ordered domain will do.

Bounds consistency infers bounds on integer, set or multiset variables. A bound on an integer variable N is any inequality, $N \leq m$ or $N \geq m$ with $\min(N) \leq m \leq \max(N)$. A bound on a set variable S is any membership constraint, $a \in S$ or $b \notin S$ with $a \in ub(S)$ and $b \in ub(S) - lb(S)$, where $lb(S)$ and $ub(S)$ are known upper and lower bounds on S wrt set containment. It may seem odd that a membership constraint is called a "bound", but such a constraint is a bound on what can or cannot be in the set. With multiset variables, this becomes even more apparent. A bound on a multiset variable M is any occurrence constraint, $occ(a, M) \leq m$ or $occ(a, M) \geq m$ with $occ(a, lb(M)) \leq m \leq occ(a, ub(M))$.

Bounds consistency infers such bounds by looking for assignments which satisfy the constraints. Such assignments must use values within the current bounds on the domains of the variables. An assignment set τ for a constraint $\mathcal{C}$ is proper, written $proper(\tau, \mathcal{C})$ iff $scope(\tau) = scope(\mathcal{C})$, for each integer variable N with $N = m \in \tau$ we have $\min(N) \leq m \leq \max(N)$, and for each set or multiset variable X with $X = a \in \tau$, we have $lb(X) \subseteq a \subseteq ub(X)$. If a proper assignment set τ for a constraint $\mathcal{C}$ satisfies that constraint, we write $\mathcal{C}(\tau)$. We write $\neg\mathcal{C}(\tau)$ otherwise.

We say that a bound on a variable is **bound inconsistent** wrt a constraint iff all proper assignment sets satisfying the bound fail to satisfy the constraint. For example, $N \geq m$ is bound inconsistent wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge N = n \in \tau \wedge n \geq m) \rightarrow \neg\mathcal{C}(\tau)$. Similarly, $a \in S$ is bound inconsistent wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge S = b \in \tau \wedge a \in b) \rightarrow \neg\mathcal{C}(\tau)$. Finally, $occ(a, M) \leq m$ is bound inconsistent wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge M = b \in \tau \wedge occ(a, b) \leq m) \rightarrow \neg\mathcal{C}(\tau)$. There is an unique **maximal** set of bound inconsistent bounds, $MaxBoundInc(\mathcal{C})$. For instance, given the constraint $|X - Y| = N$ with $\{1\} \subseteq X \subseteq \{1, 2\}$, $\{3\} \subseteq Y \subseteq \{1, 3, 4\}$ and $N = \{2, 3\}$ then $MaxBoundInc(|X - Y| = N) = \{N \geq 3, 2 \notin X, 1 \in Y\}$. The bound $N \geq 3$ is bound inconsistent as there is no proper assignment set with $N = 3$ which can satisfy $|X - Y| = N$. Similarly, $2 \notin X$ and $1 \in Y$ are bound inconsistent since the only proper assignment sets which satisfy $|X - Y| = N$ have $2 \in X$ and $1 \notin Y$.

Bounds that are **bound consistent** have at least one witness falsifying the conditions for bound inconsistency. For example, if $N \leq m$ is bound consistent wrt $\mathcal{C}$ then there is a proper assignment set τ with $N = n \in \tau$, $n \leq m$, and $\mathcal{C}(\tau)$. Similarly, if $a \notin X$ is bound consistent wrt $\mathcal{C}$ then there is a proper assignment set τ with $X = b \in \tau$, $a \notin b$, and $\mathcal{C}(\tau)$. Finally, if $occ(a, M) \geq m$ is bound consistent wrt $\mathcal{C}$ then there is a proper assignment set τ with $M = b \in \tau$, $occ(a, b) \geq m$, and $\mathcal{C}(\tau)$. For instance, given again the constraint $|X - Y| = N$ with $\{1\} \subseteq X \subseteq \{1, 2\}$, $\{3\} \subseteq Y \subseteq \{1, 3, 4\}$ and $N = \{2, 3\}$ then the bounds $N \geq 2$ and $1 \in X$ are both bound consistent since $X = \{1, 2\}$, $Y = \{3, 4\}$ and $N = 2$ satisfy the constraint and the two bounds.

The witness τ that shows that a bound is bound consistent is called a **bound support** for the bound. We extend this notion of bound support to include assignments as well as bounds. Given a constraint $\mathcal{C}$ and an integer variable N within its scope, the bound support for $N = m$ is a proper assignment set τ with $N = m \in \tau$ and $\mathcal{C}(\tau)$. Similarly, given a constraint $\mathcal{C}$ and a set or multiset variable X within its scope, the bound support for $X = a$ is a proper assignment set τ with $X = a \in \tau$ and $\mathcal{C}(\tau)$. For example, given

again the constraint $|X - Y| = N$, $X = \{1, 2\}$, $Y = \{3, 4\}$ and $N = 2$ is a bound support for $N = 2$, for $X = \{1, 2\}$ and for $Y = \{3, 4\}$.

Bound valid bounds are the dual of bound inconsistent assignments. We will need bound validity when dealing with constraint expressions involving (explicit or implicit) negation. For instance, the bound inconsistent bounds of $not(\mathcal{C})$ are exactly the bound valid bounds of $\mathcal{C}$. More formally, a bound is **bound valid** wrt a constraint iff any proper assignment set satisfying the bound must satisfy the constraint. For example, $N \geq m$ is bound valid wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge N = n \in \tau \wedge n \geq m) \rightarrow \mathcal{C}(\tau)$. Similarly, $a \in S$ is bound valid wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge S = b \in \tau \wedge a \in b) \rightarrow \mathcal{C}(\tau)$. Finally, $occ(a, M) \leq m$ is bound valid wrt $\mathcal{C}$ iff $\forall\tau.(proper(\tau, \mathcal{C}) \wedge M = b \in \tau \wedge occ(a, b) \leq m) \rightarrow \mathcal{C}(\tau)$.

As with bound inconsistency, there is an unique **maximal** set of bound valid bounds, $MaxBoundValid(\mathcal{C})$. For instance, given again the constraint $|X-Y| \neq N$ with $\{1\} \subseteq X \subseteq \{1, 2\}$, $\{3\} \subseteq Y \subseteq \{1, 3, 4\}$ and $N = \{2, 3\}$ then $MaxBoundValid(|X - Y| \neq N) = \{N \geq 3, 2 \notin X, 1 \in Y\}$. The bound $N \geq 3$ is bound valid as all proper assignment sets with $N = 3$ satisfy $|X - Y| \neq N$. Similarly, $2 \notin X$ and $1 \in Y$ are bound valid since all proper assignment sets with $2 \notin X$ or $1 \in Y$ will satisfy $|X - Y| \neq N$. Note that $|X - Y| \neq N$ is equivalent to $not(|X - Y| = N)$, and hence $MaxBoundValid(|X - Y| \neq N) = MaxBoundInc(|X - Y| = N)$.

6 Enforcing BC

When solving a constraint satisfaction problem, we can tighten the lower and upper bounds on each integer variable, prune values from the upper bound of set or multiset variables, and add values to the lower bounds until each constraint is bound consistent. More formally, a constraint $\mathcal{C}$ is **BC** (bound consistent) iff for each integer variable N in $scope(\mathcal{C})$, $N = \min(N)$ and $N = \max(N)$ have bound support, and for each set or multiset variables X in $scope(\mathcal{C})$, $lb(X)$ and $ub(X)$ are respectively the intersection and union of values assigned to X which have bound support. A set of constraints is BC iff each constraint is itself BC.

Given a bound inconsistent bound, we can prune values in the domains of the variables which are ruled out by this bound. For the bound $N \geq m$, we prune m and values larger from the domain of N. For the bound $N \leq m$, we prune m and values smaller from the domain of N. For the bound $a \in S$, we prune a from $ub(S)$. For the bound $a \notin S$, we add a to $lb(S)$. For the bound $occ(a, M) \geq m$, we reduce the number of occurrences of a in $ub(M)$ to $m - 1$. Finally, for the bound $occ(a, M) \leq m$, we increase the number of occurrences of a in $lb(M)$ to $m + 1$. Pruning all the values ruled out by the maximal set of bound inconsistent bounds will make a constraint BC or cause a domain wipeout. To return to our example with the constraint $|X - Y| = N$ where $\{1\} \subseteq X \subseteq \{1, 2\}$, $\{3\} \subseteq Y \subseteq \{1, 3, 4\}$ and $N = \{2, 3\}$. Recall that $\{N \geq 3, 2 \notin X, 1 \in Y\}$ is the maximal set of bound inconsistent bounds. Therefore we can prune 3 from N, add 2 to $lb(X)$, and delete 1 from $ub(Y)$. This gives $\{1, 2\} \subseteq X \subseteq \{1, 2\}$, $\{3\} \subseteq Y \subseteq \{3, 4\}$ and $N = \{2\}$. With these domains, the constraint $|X - Y| = N$ is BC.

To enforce BC on a constraint, we need therefore to compute the maximal set of bound inconsistent bounds. Unfortunately, this is intractable in general. We have already

proved that determining if an assignment is in $MaxInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k))$ is coNP-complete, and if a set of assignments is equivalent to $MaxInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k))$ is D^P-complete. As these proofs only used Boolean variables, we can represent the examples in these with proofs with 0/1 variables, and (in)consistency is equivalent to bounds (in)consistency. Therefore, determining if a bound is in $MaxBoundInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k))$ is coNP-complete, and if a set of bounds is equivalent to $MaxBoundInc(and(\mathcal{C}_1, \ldots, \mathcal{C}_k))$ is D^P-complete.

If each primitive constraint has an associated algorithm to compute bound inconsistent and bound valid bounds, we can instead use Inc and $Valid$ to compute sets of inconsistent and valid bounds. It is not hard to show that Inc and $Valid$ will only return bound inconsistent and bound valid bounds, and that maximality of bound inconsistency is preserved for constraint disjunction but not necessarily for constraint conjunction. We can even mix algorithms which compute bound inconsistent or bound valid bounds with those that compute inconsistent or valid assignments. When computing Inc and $Valid$, we merely need to treat the bound $M \geq m$ as short hand for the set of assignments $\{M = n \mid m \leq n \leq \max(M)\}$, and the bound $M \leq m$ as short hand for the set of assignments $\{M = n \mid \min(N) \leq n \leq m\}$. We can then compute the intersection of bounds and assignments.

7 Applications

7.1 Modeling

To show the benefits of combining constraints together with propositional connectives, we consider the orchestra rehearsal problem [13]. This is `prob039` in CSPLIB. The task is to schedule musicians in an orchestra who are rehearsing a set of pieces so that appropriate musicians are present for each piece. The model discussed in [13] uses multiple viewpoints to represent and propagate the constraints efficiently. Channelling constraints to link these viewpoints together are therefore needed. The channelling constraints take the following forms:

$$\mathit{iff}(X_i = 1, or(X_{i-1} = 1, Y_i = 1))$$

$$\mathit{iff}(Z_k = j, and(X_j = 1, X_{j-1} = 0))$$

The model also contained specialized problem specific optimality constraints of the form:

$$implies(X < Y, W_X = 1)$$

All of these constraints can be easily represented and reasoned with using the techniques described in this paper.

7.2 Valid Assignments

To propagate constraint expressions, we need algorithms for computing inconsistent and valid assignments or bounds for the primitive constraints. In the rest of this section, we show this can be done easily and effectively for a variety of constraints.

Inequalities With inequality constraints, constraint propagation only prunes bounds on the variables. To compute the maximal bound valid bounds, we can exploit the following relations: $MaxBoundValid(\mathcal{C}) = MaxBoundInc(not(\mathcal{C}))$, $not(X < Y)$ iff $X \geq Y$, $not(X \leq Y)$ iff $X > Y$, $not(X > Y)$ iff $X \leq Y$ and $not(X \geq Y)$ iff $X < Y$.

All-different The constraint $alldifferent([X_1, \ldots, X_n])$ ensures that $X_i \neq X_j$ for all $i < j$. We can use an algorithm due to Régin [12] to compute the maximal set of inconsistent assignments. It is also easy to compute the maximal set of valid assignments. Let S be the set of values that appear in the domain of more than one variable. If S is not empty, then no assignment can be valid. If S is empty, then the variable domains are disjoint and all assignments are valid.

Among The constraint $among(N, [X_1, \ldots, X_n], [d_1, \ldots, d_m])$, introduced in CHIP [2], ensures that N is equal to the number of the X_i whose values lie among the d_j. Let $a = \sum_i (domain(X_i) \subseteq \{d_1, \ldots, d_m\})$ and $b = \sum_i (|domain(X_i) \cap \{d_1, \ldots, d_m\}| > 0)$. The maximal set of valid assignments contains $N = a$ when $a = b$, and $X_i = m$ when $min(N) = max(N) = a = b$ and $m \in domain(X_i)$.

Count The constraint $count(d, [X_1, \ldots, X_m], op, N])$ where $op \in \{=, \neq, >, <, \geq, \leq\}$, introduced in SICStus Prolog [1], ensures that $|\{i \mid X_i = d\}|$ op N. For example, $count(d, [X_1, \ldots, X_m], =, N])$ ensures that N is equal to the number of variables in the X_i that take the value d. The maximal set of valid assignments can again be easily computed. For instance, given again $count(d, [X_1, \ldots, X_m], =, N])$, then the maximal set of valid assignments contains $N = a$ when $a = b$ and $X_i = m$ when $min(N) = max(N) = a = b$ and $m \in domain(X_i)$.

Element The constraint $element(I, [d_1, \ldots, d_n], X)$ ensures that $X = d_I$. This allows us to "look up" the I-th value in the array, d_j. It is again easy to compute valid assignments. The maximal set of valid assignments contains $I = j$ when $j \in domain(I)$ and $domain(X) = \{d_j\}$, and $X = m$ when $m \in domain(X)$ and for all $j \in domain(I)$ we have $d_j = m$.

Member constraint The constraint $member(N, [X_1, \ldots, X_m])$ ensures that at least one X_i takes the same value as N. It is again easy to compute valid assignments. The maximal set of valid assignments contains $N = i$ where $i \in domain(N)$ and there is some j with $domain(X_j) = \{i\}$, and $X_k = m$ where $m \in domain(X_k)$, $|domain(N)| = 1$ and there is some j with $domain(X_j) = domain(N)$, or where $m \in domain(X_k)$ and $domain(N) = \{m\}$.

Minimum and maximum The constraint $max(N, [X_1, \ldots, X_m])$ is satisfied iff N is the maximum of the values taken by the X_i, whilst $min(N, [X_1, \ldots, X_m])$ is satisfied iff N is the minimum. It is simple to see that such constraints only prune bounds, and that bound inconsistent bounds can be easily computed. Computing bound valid bounds for such constraints also involves simple bound reasoning. For example, consider $min(N, [X_1, \ldots, X_m])$. If $max(N) = min(N) = n$ and there exists i with $max(X_i) = min(X_i) = min_j(X_j) = n$ then $N \leq n$, $N \geq n$, and for any X_j, $X_j \leq k$, $X_j \geq l$ for any k or l in $range(X_j)$ are the maximal bound valid bounds. Otherwise, there are no bound valid bounds.

8 Related work

Bessière and Régin have proposed a GAC-schema like algorithm for enforcing GAC on a conjunction of primitive constraints [3]. However, as could be expected from Theorem 1, their algorithm requires time $O(d^n)$ in the worst case.

Lhomme has recently proposed an arc-consistency algorithm for disjunctions of primitive constraints $or(C_1, \ldots, C_k)$ [8]. This algorithm works by testing each assignment for membership in $\bigcap_{1 \leq i \leq k} MaxInc(C_i)$. If the assignment is not in $MaxInc(C_i)$ for some i the remaining $MaxInc$ sets do not have to be tested. However, in general $MaxInc$ for all of the primitive constraints may have to be (implicitly) computed.

The cardinality constraint can be used to implement conjunction, disjunction, negation, as well as a host of other useful constraints [5]. However, only a very restricted form of consistency is enforced on the cardinality constraint. We can demonstrate simple examples on which we can perform more pruning. For example, if C_1 is $X = 0$, C_2 is $X = 1$ and $\mathcal{D} = \{X = 0, X = 1, X = 2\}$ then $Inc(or(C_1, C_2), \mathcal{D}) = \{X = 2\}$. Pruning the value 2 from X makes the problem GAC. However, the equivalent cardinality constraint, $card(N, [C_1, C_2])$ where $N \geq 1$ is consistent without any domain prunings.

To perform more global pruning on cardinality constraints expressing disjunction, `cc(FD)` introduced constructive disjunction [6]. If any of the disjuncts, C_i in a constructive disjunction, $C_1 \vee_c \ldots \vee_c C_k$ is entailed by the constraint store, then the constructive disjunction is satisfied. Otherwise, each constraint C_i is added in turn to the constraint store and propagated. The resulting inconsistent assignments are recorded, the state restored, and the next constraint is processed. The intersection of the inconsistent assignments found for each constraint are then taken to be the inconsistent assignments for the disjunction. Constructive disjunction can do more pruning than enforcing GAC on the disjunctive constraint expression. For example, even though both $or(X = 0, Y = 0)$ and $X = Y$ on 0/1 variable are GAC, propagating $X = 0 \vee_c Y = 0$ and $X = Y$ prunes $X = Y = 1$. This extra pruning arises from interaction between the disjunctive constraint and the other constraints in the constraint store ($X = Y$ in our example). GAC on the disjunction, on the other hand, is entirely local to the disjunction. For this reason, constructive disjunction can be very expensive, and may not justify its costs in practice [7].

In contrast to these previous works, we have provided a tractable and more light weight method for computing a subset of $MaxInc$ that does a useful amount of constraint propagation. Our algorithm is compositional as it uses the propagators provided for the primitive constraints. Hence it can be applied to complex, nested logical expressions.

9 Conclusion

We have proposed a simple and light weight method for propagating complex constraint expressions built from primitive constraints using logical connectives. Since computing the maximal set of inconsistent assignments for a constraint expression is intractable in general, we have constructed a polynomial time function which computes a tractable subset compositionally. We characterised when this function is guaranteed to compute

the maximal set of inconsistent assignments. We then lifted our reasoning from inconsistent assignments to inconsistent bounds, and demonstrated how we can achieve bounds consistency on complex constraint expressions involving integer, set or multiset variables. Finally, as our approach requires being able to compute valid as well as inconsistent assignments and bounds for primitive constraints, we demonstrated that valid assignments and bounds can easily be computed for many primitive constraints. There remain many interesting directions to follow both from a theoretical and practical perspective. For example, how do we compute and use nogoods for constraint expressions? Finally, perhaps the most important question is how effective in practice will be the constraint propagation obtained from Inc?

References

1. N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. Technical report, Swedish Institute of Computer Science, 2000. SICS Technical Report T2000/01.
2. N. Beldiceanu and E. Contegean. Introducing global constraints in CHIP. *Mathematical Computer Modelling*, 20(12):97–123, 1994.
3. C. Bessière and J-C. Régin. Local consistency on conjunctions of constraints. In *Proceedings of ECAI-98 Workshop on Non-Binary Constraints*. European Conference on Artificial Intelligence, 1998.
4. C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244, 1997.
5. P. Van Hentenryck and Y. Deville. The cardinality operator: a new logical connective for constraint logic programming. In *Proceedings of the International Conference on Logic Programming (ICLP 91)*, pages 745–759, 1991.
6. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1–3):139–164, 1998.
7. J. Würtz and T. Müller. Constructive disjunction revisited. In *Proceedings of 20th German Annual Conference on Artificial Intelligence*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 377–386, Dresden, Germany, 1996. Springer-Verlag.
8. O. Lhomme. An efficient filtering algorithm for disjunction of constraints. In *Proceedings of Ninth International Conference on Principles and Practice of Constraint Programming (CP2003)*, pages 904–908. Springer, 2003.
9. C. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *Journal of Computer and System Sciences*, 28(2):244–259, 1984.
10. Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM Press, 1982.
11. Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.
12. J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on AI*, pages 362–367. American Association for Artificial Intelligence, 1994.
13. Barbara Smith. Constraint Programming in Practice: Scheduling a Rehearsal. Technical Report APES-67-2003, APES Research Group, September 2003. Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.
14. T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In F. Rossi, editor, *9th International Conference on Principles and Practices of Constraint Programming (CP-2003)*. Springer, 2003.

Optimal and Suboptimal Singleton Arc Consistency Algorithms

Christian Bessière[1] and Romuald Debruyne[2]

[1] LIRMM-CNRS, 161 rue Ada F-34392 Montpellier Cedex 5
bessiere@lirmm.fr
[2] École des Mines de Nantes, 4 rue Alfred Kastler, F-44307 Nantes Cedex 3
romuald.debruyne@emn.fr

Abstract. Singleton arc consistency (SAC) is a local consistency that ensures that a constraint network can be made arc consistent after any assignment of a value to a variable. A first contribution of this paper is to show experimentally that the optimal-time algorithm proposed in [5] (which was analyzed only theoretically) is efficient in practice compared to previous SAC algorithms. However, it can be costly in space on large problems, even with the small improvements we propose at the beginning of this paper. To reduce this space consumption, we propose another SAC algorithm requiring less space but no longer optimal in time. An experimental study on random problems highlights the good performance of this second algorithm.

1 Introduction

Ensuring that a given local consistency does not lead to a failure when we enforce it after having assigned a variable to a value is a common idea in constraint solving. It has been applied (sometimes under the name 'shaving') in constraint problems with numerical domains by limiting the assignments to bounds in the domains and ensuring that bounds consistency does not fail [13]. In SAT, it has been used as a way to compute more accurate heuristics for DPLL [10, 14]. Finally, in constraint satisfaction problems (CSPs), it has been proposed and studied under the name Singleton Arc Consistency (SAC) ([9, 19]).

Some nice properties give to SAC a real advantage over the other local consistencies enhancing the ubiquitous arc consistency. Its definition is much simpler than restricted path consistency [3], max-restricted-path consistency [8], or other exotic local consistencies, and its operational semantics can be understood by a non-completely-expert of the field. Enforcing it only removes values in domains, and thus does not change the structure of the problem, as opposed to path consistency [17], k-consistency [11], etc. Finally, implementing it can be done simply on top of any AC algorithm.

Non optimal SAC algorithms were proposed in [9] and [1] while an optimal one has recently been described in [5]. In Section 3 of this paper, we rewrite the algorithm proposed in [5] in a slight different way that does not change its worst-case time and space complexities while improving slightly its practical

performance. We call it SAC-Opt. However, the optimal time complexity is kept at the cost of a high space complexity that prevents the use of this algorithm on large problems. We then propose in Section 4 another SAC algorithm, SAC-SDS, with a better worst-case space complexity but no longer optimal in time. Nevertheless, its time complexity remains better than the other SAC algorithms proposed in the past. The experiments presented in Section 5 highlight the good performance of both SAC-Opt and SAC-SDS.

2 Preliminaries

A finite *constraint network* P consists of a finite set of n variables $X = \{i, j, \ldots\}$, a set of domains $D = \{D_i, D_j, \ldots\}$, where the domain D_i is the finite set of values that variable i can take, and a set of constraints $C = \{c_1, \ldots, c_m\}$. Each constraint c_i is defined by the ordered set $var(c_i)$ of the variables it involves, and a set $sol(c_i)$ of allowed combinations of values. An assignment of values to the variables in $var(c_i)$ *satisfies* c_i if it belongs to $sol(c_i)$. A *solution* to a constraint network is an assignment of a value from its domain to each variable such that every constraint in the network is satisfied. We will use c_{ij} to refer to $sol(c)$ when $var(c) = (i, j)$. $\Phi(P)$ denotes the network obtained after enforcing Φ-consistency on P.

Definition 1 *A constraint network $P = (X, D, C)$ is said to be* ***Φ-inconsistent*** *iff $\Phi(P)$ has some empty domains or empty constraints.*

Definition 2 *A constraint network $P = (X, D, C)$ is* ***singleton arc consistent*** *iff $\forall i \in X, \forall a \in D_i$, the network $P|_{i=a}$ obtained by replacing D_i by the singleton $\{a\}$ is not arc inconsistent.*

3 An Optimal Algorithm for SAC

SAC-1 [9] has no data structure storing on which values rely the SAC consistency of each value. After a value removal, SAC-1 must check again the SAC consistency of all the other values.

SAC-2 [1] uses the fact that if we know that AC does not lead to a wipe out in $P|_{i=a}$ then the SAC consistency of (i, a) holds as long as all the values in $AC(P|i = a)$ are in the domain. After the removal of a value (j, b), SAC-2 checks again the SAC consistency of all the values (i, a) such that (j, b) was in $AC(P|_{i=a})$. This leads to a better average time complexity than SAC-1 but the data structures of SAC-2 are not sufficient to reach optimality since SAC-2 may waste time redoing the enforcement of AC in $P|_{i=a}$ several times from scratch.

Algorithm 1, called SAC-Opt, is an algorithm that enforces SAC in $O(end^3)$, the lowest time complexity which can be expected. (See [5]).

The idea behind such an optimal algorithm is that we don't want to do and redo (potentially nd times) arc consistency from scratch for each subproblem $P|_{j=b}$ each time a value (i, a) is found SAC inconsistent. (Which represents

Algorithm 1: `The optimal SAC algorithm`

```
procedure SAC-Opt(in P:Problem);
    /* init phase */;
 1  P ← AC(P) ; PendingList ← ∅;
    foreach (i, a) ∈ D do P_ia ← nil;
 2  foreach (i, a) ∈ D do
 3      P_ia ← P /* we copy the network and its data structures */;
 4      if not(propagateAC(P_ia, D_i \ {a})) then
 5          D ← D \ {(i, a)};
 6          propagateAC(P, {(i, a)});
            foreach P_jb ≠ nil such that (i, a) ∈ P_jb do
 7              Q_jb ← Q_jb ∪ {(i, a)};
 8              PendingList ← PendingList ∪ P_jb;
    /* propag phase */;
 9  while PendingList ≠ ∅ do
        pop P_ia from PendingList;
10      if not(propagateAC(P_ia, Q_ia)) then
11          D ← D \ {(i, a)};
            foreach (j, b) ∈ D such that (i, a) ∈ P_jb do
12              Q_jb ← Q_jb ∪ {(i, a)};
13              PendingList ← PendingList ∪ P_jb;
```

n^2d^2 potential arc consistency calls.) To avoid such costly repetitions of arc consistency calls, we duplicate the problem nd times, one for each value (i, a), so that we can benefit from the incrementality of arc consistency on each of them. An AC algorithm is called 'incremental' when its complexity on a problem P is the same for a single call or for up to nd calls, where two consecutive calls differ only by the deletion of some values from P. The generic AC algorithms are all incremental.

SAC-Opt can be decomposed in several sequential steps. In the following, `propagateAC`(P, S) denotes the function that incrementally propagates in P the removal of the set S of values when an initial AC call has already been executed, initializing the data structures required by the AC algorithm in use.

First, after some basic initializations and making the problem arc consistent (line 1), the loop in line 2 duplicates nd times the arc consistent problem obtained in line 1, and propagates the removal of all the values different from a for i in each $P|_{i=a}$, denoted P_{ia} (line 4). If a subproblem P_{ia} has no arc consistent subdomain, the removal of (i, a) is propagated in P (line 6). The subproblems corresponding to the subsequent steps of the loop will benefit from this propagation because they are created by duplication of P (line 3).[3] For each already

[3] This is the main difference between SAC-Opt and the algorithm presented in [5]. SAC-Opt will thus build *less* and *smaller* subproblems.

checked subproblem P_{jb} having (i, a) in its domain, (i, a) is put in Q_{jb} for future propagation (line 7).

Once this initialization phase has finished, the removal of all SAC inconsistent value has been propagated in all the subproblems except in those in *PendingList*. Each problem P_{jb} in *PendingList* contains the removals that must be propagated in its local propagation list Q_{jb}. During the whole loop of the propagation phase, if the AC propagation of a list Q_{ia} in a subproblem P_{ia} fails (line 10), (i, a) is removed from D, and the list Q_{jb} of each subproblem P_{jb} having (i, a) in its domain is updated for a future propagation of this removal.

When *PendingList* is empty, all the removals have been propagated in the subproblems and all the values in D are SAC consistent.

Theorem 1 *SAC-Opt is a correct SAC algorithm with $O(end^3)$ optimal worst-case time complexity and $O(end^2)$ worst-case space complexity.*

Proof. (See [5].)

Remarks. We can remark that the Q lists contain values to be propagated. This is written like this because the AC algorithm chosen is not specified here, and value removal is the most accurate information we can have. If the AC algorithm chosen is AC-6 [4], AC-7 [6] , or AC-4 [16], the lists will be directly used like this. If it is AC-3 [15] or AC-2001 [7], only the variables from which the domain has changed are necessary. This last information is trivially obtained from the list of removed values.

We can also point out that if AC-3 is used, we decrease the space complexity to $O(n^2d^2)$, but time complexity increases to $O(end^4)$ since AC-3 is not optimal.

4 Losing Time Optimality to Save Space

SAC-Opt cannot be used on large constraint networks because of its $O(end^2)$ space complexity. Moreover, it seems difficult to reach optimal time complexity with smaller space requirements. Indeed, a SAC algorithm has to enforce AC in each subproblem $P|_{i=a}$ and to be optimal in time it must store sufficient data to never redo some work in a subproblem. Optimal AC algorithms use at least a space in $O(ed)$ and it seems therefore unavoidable that an optimal time SAC algorithm requires nd times more space.

In this section we propose to relax time optimality to reach a satisfactory trade-off between space and time. To avoid a too general discussion, we instantiate this idea on a AC-2001 oriented SAC algorithm. The "suboptimal" algorithm we present uses AC-2001 data structures, but the same idea could be implemented with other low-space optimal AC algorithms such as AC-6 and AC-7. The algorithm SAC-SDS ('Sharing Data Structures') tries to use the incrementality of the AC algorithms to avoid redundant work, without duplicating on each subproblem $P|_{i=a}$ the data structures required by optimal AC algorithms. This algorithm requires less space than SAC-Opt but is not optimal in time.

Algorithm 2: `The SAC-SDS algorithm`

```
procedure SAC-SDS-2001(in P: Problem);
1  P ← AC-2001(P) ; PendingList ← ∅;
2  foreach (i, a) ∈ D do
3      Support_ia^SAC ← nil ; Q_ia ← {i}; PendingList ← PendingList ∪ {(i, a)};
4  while PendingList ≠ ∅ do
       pop (i, a) from PendingList ;
       if a ∈ D_i then
5          if Support_ia^SAC = nil then Support_ia^SAC ← (D \ D_i) ∪ {(i, a)};
6          if not( propagateSubAC( (X, Support_ia^SAC, C), Q_ia)) then
7              D_i ← D_i \ {a} ;
8              updateSubproblems((i, a)) ;
9              propagateAC( (X, D, C), {i}) ;

function propagateAC(in (X, D, C): Problem, in Q: set): Boolean;
   while Q ≠ ∅ do
       pop j from Q ;
       foreach i ∈ X such that ∃C_ij ∈ C do
           foreach a ∈ D_i such that Last_ija ∉ D_j do
10             if ∃b ∈ D_j such that b > Last_ija ∧ C_ij(a, b) then
11                 Last_ija ← b /* not in propagateSubAC */;
               else
12                 D_i ← D_i \ {a} ;
                   Q ← Q ∪ {i} ;
13                 updateSubproblems((i, a)) /* not in propagateSubAC */;
           if D_i = ∅ then return false;
   return true;
procedure updateSubproblems(in (i, a): Value);
   foreach (j, b) ∈ D such that (i, a) ∈ Support_jb^SAC do
14     Support_jb^SAC ← Support_jb^SAC \ {(i, a)} ;
       Q_jb ← Q_jb ∪ {i} ;
15     PendingList ← PendingList ∪ {(j, b)} ;
```

The main idea in SAC-SDS is that for each value (i, a), we store a local propagation list and the domain of $AC(P|_{i=a})$, denoted by $Support_{ia}^{SAC}$ and called its SAC-support. Thanks to these SAC-supports, we know which values may no longer be SAC consistent after a removal. These SAC-supports are also used to follow the AC enforcement in each subproblem $P|_{i=a}$ with the domains in the state in which they were at the end of the last AC propagation.

SAC-SDS-2001 relies on AC-2001. The data structure *Last* of this algorithm will be used for the propagation of AC in P but it will also be used to help the enforcement of AC in the subproblems $P|_{i=a}$. This data structure is therefore shared since it is not duplicated while being used for achieving AC in P and all the subproblems $P|_{i=a}$.

In SAC-SDS-2001, a value (i, a) is in *PendingList* if some removals have to be propagated in $P|_{i=a}$. In such a case, Q_{ia} is a non empty list composed of all the variables j such that values in D_j have been removed since the last AC enforcement in $P|_{i=a}$. After some initializations, SAC-SDS-2001 repeatedly pops a value from *PendingList* and propagates AC in $(X, Support_{ia}^{SAC}, C)$, namely $P|_{i=a}$, since $Support_{ia}^{SAC}$ is the current domain of $P|_{i=a}$. Remark that if $Support_{ia}^{SAC} = nil$ this is the first enforcement of AC in $P|_{i=a}$ and $Support_{ia}^{SAC}$ must be initialized (line 5). If $P|_{i=a}$ is arc inconsistent, (i, a) is not SAC consistent. It is therefore removed from P (line 7) and from the subproblems (using *updateSubproblems* in line 8) before the propagation of this removal (line 9).

The function *propagateSubAC* used to propagate arc consistency in the subproblems is almost similar to *propagateAC* in AC-2001. The difference comes from line 11 where the structure *Last* is not updated. Indeed, this data structure is useful to achieve AC in the subproblems more quickly (line 10) since we know that there is no support for (i, a) on C_{ij} lower than $Last_{ija}$ in P (and so in subproblems) but since this data structure is not duplicated for each subproblem it must not be updated by *propagateSubAC*.

Obviously, while achieving AC in P using *propagateAC* the data structure *Last* is updated and the only difference with AC-2001 is the line 13 where *updateSubproblems* is used to remove the SAC inconsistent value (i, a) in all the subproblems and to update the local propagation lists for future propagation of these removals.

By using *updateSubproblems*, SAC-SDS-2001 tries to avoid redoing the same propagations in all the subproblems. Each removal of a SAC inconsistent value is first propagated in P before being propagated in the subproblems. Thanks to *updateSubproblems*, all the subproblems will benefit from the removals in P.

Theorem 2 *SAC-SDS is a correct SAC algorithm with* $O(end^4)$ *time complexity and* $O(n^2d^2)$ *space complexity.*

Proof. Correctness. Note first that the structure *Last* is updated only while achieving AC in P so that any support of (i, a) in D_j is greater than or equal to $Last_{ija}$. The domains of the subproblems being subdomains of D, any support of a value (i, a) on C_{ij} in a subproblem is also greater or equal to $Last_{ija}$. This explains that *propagateSubAC* can benefit (line 10) from the structure *Last* without losing any support.

Suppose that SAC-SDS-2001 is not sound on a problem P and let (i, a) be the first SAC consistent value it removes while it should not. If (i, a) is removed at line 7 it would be a SAC inconsistent value since only deleted values are put in the local propagation lists and by assumption any previously removed value is SAC-inconsistent. So, (i, a) is removed at line 9 because it is no longer arc consistent after the removal of some SAC inconsistent values and (i, a) is therefore not SAC consistent. So, any removed value is SAC inconsistent and SAC-SDS-2001 is sound.

Completeness comes from the fact that any deletion is propagated. After the initialization (lines 2-3), $PendingList = D$ and so, the main loop of SAC-SDS-2001 considers any subproblem $P|_{i=a}$ at least once. Each time a value (i, a)

is found SAC inconsistent in P, because $P|_{i=a}$ is arc inconsistent (line 6) or because the deletion of some SAC-inconsistent values make it arc inconsistent in P (lines 9 and 12 of *propagateAC*), (i, a) is removed from the subproblems (using *updateSubproblems* and *PendingList*) and the local propagation lists are updated for future propagation. At the end of the main loop, *PendingList* is empty, so all the removals have been propagated and for any value $(i, a) \in D$ $Support_{ia}^{SAC}$ is a non empty arc consistent subdomain of $P|_{i=a}$.

Complexity. The data structure *Last* requires a space in $O(ed)$. Each of the nd $Support_{ia}^{SAC}$ can contain nd values and there is at most n variables in the nd local propagation lists. Since $e < n^2$, the space complexity of SAC-SDS-2001 is in $O(n^2d^2)$. So, considering space requirements, SAC-SDS-2001 is similar to SAC-2 [1].

Regarding time complexity, SAC-SDS-2001 first duplicates the data structures and propagates arc consistency on each subproblem (lines 5 and 6), two tasks which are respectively in $nd \cdot nd$ and $nd \cdot ed^2$. Each value removal is propagated to all $P|_{i=a}$ problems via an update of *PendingList* and $Support_{ia}^{SAC}$ sets (lines 8 and 13). This requires $nd \cdot nd$ operations. Each subproblem can in the worst case be called nd times for arc consistency, and there are nd subproblems. The domains of each subproblem are stored so that the AC propagation is launched with the domains in the state in which they were at the end of the previous AC propagation in the subproblem. Thus, in spite of the several AC propagations on a subproblem, a value will be removed at most once and, thanks to incrementality of arc consistency, the propagation of these nd value removals is in $O(ed^3)$. (Remark that we cannot reach the optimal ed^2 complexity for arc consistency on these subproblems since we do not duplicate the data structures necessary for AC optimality.) Thus the total cost of arc consistency propagations is $nd \cdot ed^3$. The total time complexity is $O(end^4)$. □

As SAC-2, SAC-SDS performs a better propagation than SAC-1 since after the removal of a value (i, a) from D, SAC-SDS checks the arc consistency of the subproblems $P|_{j=b}$ only if they have (i, a) in their domains (and not all the subproblems as SAC-1). But this is not sufficient to have a better worst-case time complexity than SAC-1. The time complexity of SAC-2 is indeed $O(en^2d^4)$ as SAC-1. SAC-SDS improves this complexity because it does not propagate in the subproblems from scratch since the *current* domain of each subproblem is stored (using $Support^{SAC}$). Furthermore, we can expect a better average time complexity since the shared structure *Last* can reduce the number of arc consistency tests required. Finally, SAC-SDS does not duplicate data structures to test the arc consistency of a subproblem, so, no restoration of data structures is required after such a test.

5 Experimental Results

To compare the performances of the SAC algorithms, we used the random uniform constraint network generator of [12] which produces instances according to the Model B [18]. All the algorithms have been implemented in C++. SAC-1 and

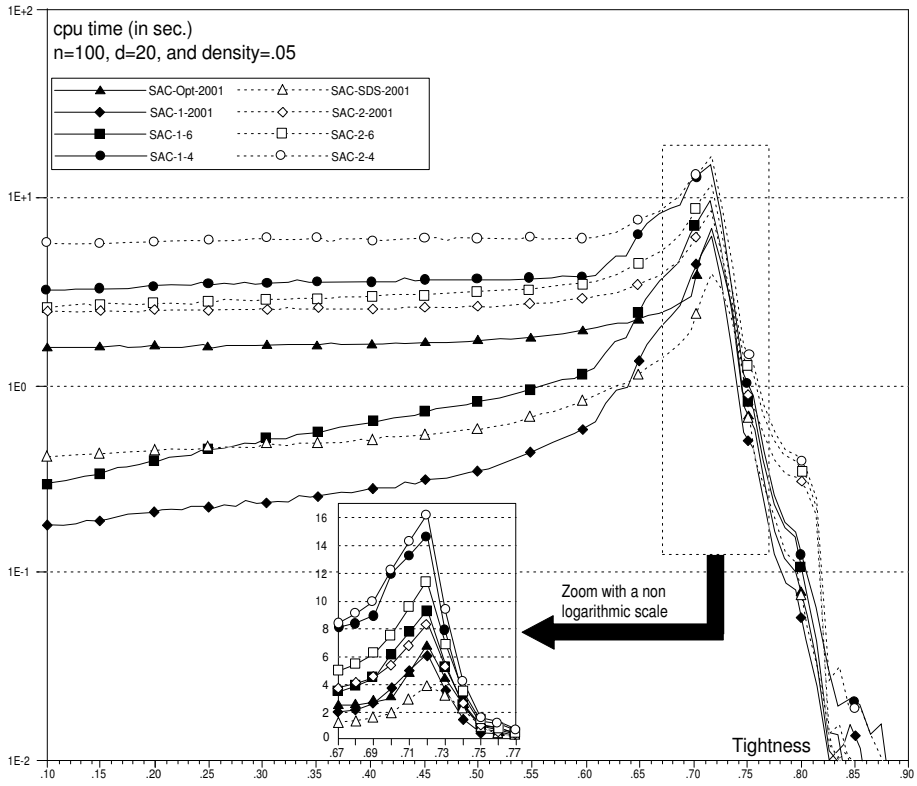


Fig. 1. cpu time results on constraint networks with n=100, d=20, and density=.05.

SAC-2 have been tested using several AC algorithms. In the following, we note AC-1-X, AC-2-X and SAC-Opt-X the versions of SAC-1, SAC-2 and SAC-Opt based on AC-X. Note that for SAC-2 the implementation of the propagation list has been done according to the recommendations made in [2, 1].

5.1 Experiments on sparse constraint networks

Fig. 1 presents cpu time performances on constraint networks having 100 variables, 20 values in each initial domain, and a density of .05. These constraint networks are relatively sparse since the variables have five neighbors on average. For each tightness, 50 instances were generated. Fig. 1 shows mean values obtained on a Pentium IV-1600 MHz with 512 Mb of memory under Windows XP.

For a tightness lower than .55, all the values are SAC consistent. On these under constrained network, the SAC algorithms check the arc consistency of each subproblem at most once. Storing the SAC-supports to enhance the propagation, as in SAC-2 and in SAC-SDS-2001, does not pay-off on these problems.

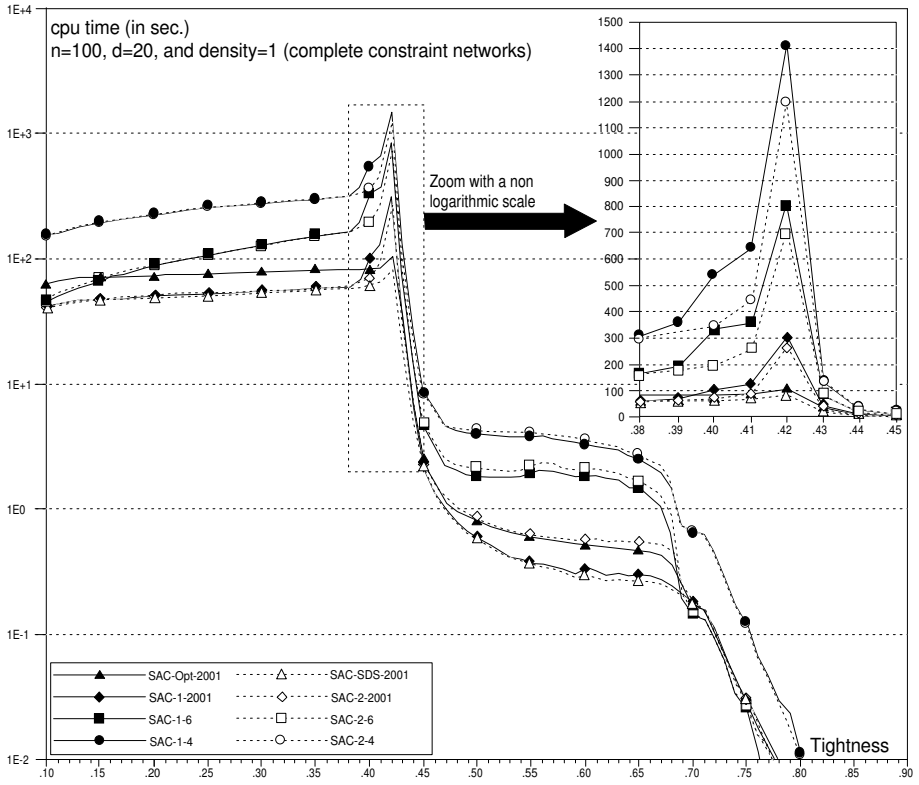


Fig. 2. cpu time results on complete constraint networks with n=100 and d=20.

A brute-force algorithm such as SAC-1 is sufficient. SAC-1-2001 shows the best performance.

On problems having tighter constraints, some SAC inconsistent values are removed and at tightness .72 we can see a peak of complexity. However, as mentioned in [2], the enhanced propagation of SAC-2 is useless on sparse constraint networks and SAC-2-X (with X$\in \{4, 6, 2001\}$) is always more expensive than SAC-1-X on the generated problems.

Around the peak of complexity, SAC-SDS-2001 is the clear winner. SAC-Opt-2001 and SAC-1-2001 are around 1.7 times slower, and all the others are between 2.1 and 10 times slower.

5.2 Experiments on dense constraint networks

We used the same computer to evaluate the performance of the SAC algorithms on complete constraint networks. For each tightness, 50 instances were generated and Fig. 2 shows mean values.

The performance of SAC-2 and SAC-1 is very close. When all the values are SAC consistent (tightness lower than .37) the additional data structure of SAC-2

is useless since there is no propagation. However the cost of building this data structure is not important compared to the overall time and the time required by SAC-2 is almost the same as SAC-1. Around the peak of complexity, SAC-2-X (with X$\in \{4, 6, 2001\}$) requires a little less time than SAC-1-X. SAC-2 has to repeatedly recheck the arc consistency of less subproblems than SAC-1 but the cost of testing a subproblem remains the same. On very tight constraints, SAC-1 requires less time than SAC-2 since the inconsistency of the problem is found with almost no propagation and building the data structure of SAC-2 is useless.

Conversely to what is supposed in [1], using AC-4 in SAC-1 (or in SAC-2) instead of AC-6 or AC-2001 is not worthwhile. The intuition was that since the data structure of AC-4 has not to be updated, the cost of its creation would be light compared to the profit we can expect. However, SAC-1-4 and SAC-2-4 are far more costly than their versions based on AC-6 or AC-2001.

The best results are obtained with SAC-Opt-2001 and SAC-SDS-2001 which are between 2.6 and 17 times faster than the others at the peak. These two algorithms have a better propagation between subproblems than SAC-1 but they also avoid some redundant work and so reduce the work performed on each subproblem.

6 Summary and Conclusion

We have presented SAC-Opt, a slightly modified version of the optimal worst-case time complexity SAC algorithm presented in [5]. However, the $O(end^2)$ space complexity of this algorithm prevents its use on large constraint networks. Therefore, we have proposed another SAC algorithm, SAC-SDS, that is not optimal in time but that requires less space than SAC-Opt. Like the optimal algorithm, SAC-SDS tries to avoid redundant work. Experiments show the good performance of these new SAC algorithms.

References

1. R. Barták. A new algorithm for singleton arc consistency. In *Proceedings FLAIRS'04*, Miami Beach, Fl, 2004. AAAI Press.
2. R. Barták and R. Erben. Singleton arc consistency revised. In *ITI Series 2003-153*, Prague, 2003.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE-CAIA'95*, Los Angeles, CA, 1995.
4. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence 65*, pages 179–190, 1994.
5. C. Bessiere and R. Debruyne. Theoretical analysis of singleton arc consistency. In B. Hnich, editor, *Proceedings ECAI'04 workshop on Modelling and Solving Problems with Constraints*, Valencia, Spain, 2004.
6. C. Bessière, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.
7. C. Bessière and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle, WA, 2001.

8. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings CP'97*, pages 312–326, Linz, Austria, 1997.
9. R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings IJCAI'97*, pages 412–417, Nagoya, Japan, 1997.
10. J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.
11. E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.
12. D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random uniform csp generators. In *http://www.ics.uci.edu/˜ frost/csp/generatotr.html*, 1996.
13. O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI'93*, pages 232–238, Chambéry, France, 1993.
14. C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, 1997.
15. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
16. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
17. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.
18. P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
19. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings CP'00*, pages 353–368, Singapore, 2000.

Revision ordering heuristics for the Constraint Satisfaction Problem

Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France
{boussemart,hemery,lecoutre}@cril.univ-artois.fr

Abstract. For many years, arc consistency has been recognized as a basic property of constraint networks. Among all algorithms that have been proposed to establish arc consistency, AC3, and more precisely its recent extensions, still remains competitive. In this paper, we present three variants for AC3 based algorithms and we focus on the order in which revisions are applied by them. For the three variants, which respectively correspond to algorithms with an arc-oriented, variable-oriented and constraint-oriented propagation scheme, we propose some original revision ordering heuristics and adapt the ones defined in [21]. The experimentation which has been run both on binary and non-binary problems confirm that using such heuristics, when arc consistency is used as a preprocessing or/and when it is maintained during search, turns out to significantly reduce the number of constraint checks. Furthermore, we show that the variable-oriented variant is guaranteed to benefit from such heuristics in terms of cpu time.

1 Introduction

For many years, arc consistency has been recognized as a basic property of constraint networks. Arc consistency guarantees that any value of the domain of a variable can be found in, at least, a support of any constraint. This property can be established by an algorithm to make a constraint network arc consistent but it can also be maintained during the search of a solution.

Many algorithms have been proposed to establish arc consistency. One of the very first proposals is the algorithm AC3 [12]. This coarse grained algorithm involves applying successive revisions of arcs, i.e., of pairs (C, X) composed of a constraint C and of a variable X belonging to the set of variables of C. Later, other algorithms such as AC4 [14], AC6 [2] and AC7 [3] have been introduced. These fine grained algorithms involve applying successive revisions of "values", i.e., of triplets (C, X, a) composed of an arc (C, X) and of a value a belonging to the domain of X.

Even if, theoretically, AC3, unlike AC4, AC6 and AC7, has not an optimal worst case time complexity ($O(md^3)$ for AC3 and $O(md^2)$ for AC4, AC6 and

AC7 where m and d respectively denote the number of constraints and the size of the uniform domains) and if, in practice, AC3 is not always as fast as AC6 and AC7, AC3 has the great advantage to be easily implemented.

On the other hand, some recent extensions of AC3 have been developed which, while preserving the simplicity of AC3, turn out to be as competitive as fine grained algorithms (with, in particular, a worst case time complexity in $O(md^2)$ for most of them). These new algorithms are called AC2000 [5], AC2001/3.1 [5, 22], AC3$_d$ [18], AC3.2 [11] and AC3.3 [11]. It is also interesting to note that with respect to some desirable properties of arc consistency algorithms, it is possible to draw a parallel [5] between AC2001/3.1 and AC6 and another [11] between AC3.3 and AC7. Then, it clearly appears that AC3 based algorithms are up-to-date algorithms to establish arc consistency.

In this paper, we are interested in the order in which revisions are applied by AC3-based algorithms. First, we present three variants which respectively correspond to algorithms with an arc-oriented, variable-oriented and constraint-oriented propagation scheme. The first one is the most commonly presented, the second one corresponds to the algorithm proposed by [13, 6], and the third one is original. Even if, at first glance, all these variants seems to be equivalent, we shall emphasize a significant difference of their respective behaviour when considering different revision ordering heuristics. As far as we are aware, the only works which concern such heuristics are [21, 8, 18] which focus on the arc-oriented variant to solve binary problems, and [15] which focuses on fine-grained algorithms.

Then, our contribution is the study of various revision ordering heuristics with respect to the use of the three variants before and/or during the search of a solution for binary and non binary problems. In particular, we propose some original heuristics based on the proportion of removed values in the different domains and on the current domain size of the different constraints. Also, we adapt to the three proposed variants, the heuristic of [21] which is based on the current size of the domains. Experimental results show that using such heuristics before or/and during the search of a solution can be quite efficient in terms of constraint checks. Furthermore, we show that the variable-oriented variant is guaranteed to benefit from such heuristics in terms of cpu time.

This paper is organized as follows. Section 2 introduces some preliminaries. In Section 3, three variants of the basic arc consistency algorithm are described. Some revision ordering heuristics are presented in Section 4. Before concluding, Section 5 gives an experimental evaluation.

2 Preliminaries

Let us introduce some notations frequently used in the rest of the paper:

- $|S|$ denotes the cardinality of a set S, i.e. the number of elements of S,
- $\prod_{i=1}^{k} S_i$ denotes the Cartesian product over k sets $S_1, \ldots, S_k$, i.e. the set $\{(a_1, \ldots, a_k) \,|\, a_i \in S_i, 1 \leq i \leq k\}$,

Definition 1. *A constraint network is a pair* $(\mathscr{X}, \mathscr{C})$ *where:*

- $\mathscr{X} = \{X_1, \ldots, X_n\}$ *is a finite set of* n *variables such that each variable* X_i *has an associated domain* $dom(X_i)$ *denoting the set of values allowed for* X_i*,*
- $\mathscr{C} = \{C_1, \ldots, C_m\}$ *is a finite set of* m *constraints such that each constraint* C_j *has an associated relation* $rel(C_j)$ *denoting the set of tuples allowed for the variables* $vars(C_j)$ *involved in the constraint* C_j*.*

We shall say that a constraint C involves (or binds) a variable X if and only if X belongs to $vars(C)$. The arity of a constraint C is the number of variables involved in C, i.e., the number of variables in $vars(C)$. A binary constraint only involves 2 variables. The domain of a constraint C, denoted $\mathrm{dom}(C)$, is the Cartesian product $\prod_{j=1}^{k} dom(X_{i_j})$, where C is a k-ary constraint such that $\mathrm{vars}(C) = \{X_{i_1}, \ldots, X_{i_k}\}$. For any element $t = (a_{i_1}, \ldots, a_{i_k})$, called k-tuple, of $\mathrm{dom}(C)$, $t[X_{i_j}]$ denotes the value a_{i_j}. A k-tuple t is said to be a support of C iff $t \in rel(C)$ and is said to be a support of (X,a) in C iff t is a support of C such that $t[X] = a$. Determining if a tuple is allowed by a constraint C (i.e. is a support of C) is called a constraint check.

An instance of the Constraint Satisfaction Problem (CSP) is defined by a constraint network. A CSP instance is said to be satisfiable iff the constraint network to which it corresponds admits a solution, and unsatisfiable otherwise. Solving a CSP instance involves either finding one (or more) solution or determining its unsatisfiability. A solution is an assignment of values to all the variables such that all the constraints are satisfied.

To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Usually, constraint propagation algorithms, which are based on some constraint network properties such as arc-consistency, remove some values which can not occur in any solution. Modifying the domains of a given constraint network in order to get it arc consistent involves performing constraint checks.

Definition 2. *Let* $P = (\mathscr{V}, \mathscr{C})$ *be a CSP,* $C \in \mathscr{C}$*,* $V \in vars(C)$ *and* $a \in dom(V)$*.* (V, a) *is said to be consistent wrt* C *iff there exists a support of* (V, a) *in* C*.* C *is said to be arc-consistent iff* $\forall V \in vars(C)$*,* $\forall a \in dom(V)$*,* (V, a) *is consistent wrt* C*.* P *is said to be arc consistent iff* $\forall C \in \mathscr{C}$*,* C *is arc-consistent.*

3 AC3 based algorithms

In this section, we present the basic coarse-grained algorithm to establish arc consistency, namely, AC3 [12]. Even if fine-grained algorithms such as AC4 [14], AC6 [2] and AC7 [3] have been introduced, the simplicity and relative efficiency (e.g., [20]) of AC3 have contributed to the fact that it is not still out of date. Further, some recent improvements, which are quite competitive with fine-grained algorithms, have been proposed. These new coarse-grained algorithms (or AC3 based algorithms) respectively correspond to AC2000 [5], AC2001/3.1 [5, 22],

Algorithm 1 arc-oriented AC3

```
1: Q ← {(C, X) | C ∈ 𝒞 ∧ X ∈ vars(C)}
2: while Q ≠ ∅ do
3:    pick (C, X) in Q
4:    nbRemovals ← revise(C, X)
5:    if nbRemovals > 0 then
6:       if dom(X) = ∅ then return FAILURE
7:       else Q ← Q ∪ {(C', X') | X ∈ vars(C') ∧ X' ∈ vars(C') ∧ X ≠ X' ∧ C ≠ C'}
8:    end if
9: end while
10: return SUCCESS
```

Algorithm 2 revise(C : Constraint, X : Variable) : integer

```
1: nbRemovals ← 0
2: for each a ∈ dom(X) do
3:    if seekSupport(C, X, a) = false then
4:       remove a from dom(X)
5:       nbRemovals ← nbRemovals + 1
6:    end if
7: end for
8: return nbRemovals
```

$AC3_d$ [18], AC3.2 [11] and AC3.3 [11]. Even if, for the sake of simplicity, from now on, we shall be mainly concerned with AC3, all what follows can be adapted to other coarse-grained algorithms.

As already stated, AC3 is a coarse-grained algorithm, that is to say, an algorithm whose principle is to apply successive revisions of pairs (C, X), called arcs, composed of a constraint C and of a variable X belonging to the set of variables of C. Each revision of an arc (C, X) aims at removing the values of $dom(X)$ without any support in C.

AC3 requires the management of a set Q recording the revisions to be still performed. Basically, Q corresponds to a set of arcs [12]. However, it is possible to consider Q as a set of variables [13, 6, 5, 22], and also, as a set of constraints. We present these three alternatives in the general context of non-binary constraint networks.

3.1 Arc-oriented AC3

First, we describe the variant of AC3 which uses an arc-oriented propagation scheme. This variant, which is simple, natural and the most commonly presented, is depicted in Algorithm 1. Initially, all arcs (C, X) are put in a set Q. Then, each arc is revised in turn, and when a revision is effective (at least one value has been removed), the set Q has to be updated. A revision is performed by a call to the function $revise(C, X)$, depicted in Algorithm 2, and removes values of dom(X) that have become inconsistent with respect to C. This function returns the number of removed values. On the other hand, the function *seekSupport*

Algorithm 3 variable-oriented AC3

```
1: Q ← {X | X ∈ 𝒳}
2: ∀C ∈ 𝒞, ∀X ∈ vars(C), ctr(C, X) ← 1
3: while Q ≠ ∅ do
4:    pick X in Q
5:    for each C | X ∈ vars(C) do
6:       if ctr(C, X) = 0 then continue
7:       for each Y ∈ vars(C) do
8:          if needsNotBeRevised(C, Y) then continue
9:          nbRemovals ← revise(C, Y)
10:         if nbRemovals > 0 then
11:            if dom(Y) = ∅ then return FAILURE
12:            Q ← Q ∪ {Y}
13:            for each C' | C' ≠ C ∧ Y ∈ vars(C') do
14:               ctr(C', Y) ← ctr(C', Y) + nbRemovals
15:            end for
16:         end if
17:      end for
18:      for each Y ∈ vars(C) do ctr(C, Y) ← 0
19:   end for
20: end while
21: return SUCCESS
```

Algorithm 4 needsNotBeRevised(C : Constraint, X : Variable) : boolean

```
1: return (ctr(C, X) > 0 and ∄Y ∈ vars(C) | Y ≠ X ∧ ctr(C, Y) > 0)
```

determines whether there exists a support of (X, a) in C. According to the the implementation of this function, we obtain the different AC3 based algorithms. The algorithm is stopped when the set Q becomes empty.

3.2 Variable-oriented AC3

The second variant of AC3, uses a variable-oriented propagation scheme as proposed by [13, 6]. The principle is to insert in Q all variables with reduced domains. Initially, all variables are inserted in Q. Then, iteratively, each variable X of Q is selected and each constraint C binding X is considered. Then, it is possible to perform the revision of all arcs (C, Y) with $Y \neq X$. When the revision of an arc (C, Y) involves the removal of some values in $dom(Y)$, the variable Y is added to Q.

The reader can notice that the description of Algorithm 3 differs slightly from the principle presented just above. Indeed, to avoid useless treatments, it is necessary to introduce some counters in order to determine whether a given revision is essential. For instance, let us assume a binary constraint $C_{i,j}$ binding the variables X_i and X_j. If the selection of the variable X_i involves an effective revision of $(C_{i,j}, X_j)$ (i.e., the removal of, at least, a value from the domain of X_j), and, if next, the selection of X_j involves an effective revision of $(C_{i,j}, X_i)$,

Algorithm 5 constraint-oriented AC3

```
1: Q ← {C | C ∈ 𝒞}
2: ∀C ∈ 𝒞, ∀X ∈ vars(C), ctr(C, X) ← 1
3: while Q ≠ ∅ do
4:    pick C in Q
5:    for each Y ∈ vars(C) do
6:       if needsNotBeRevised(C, Y) then continue
7:       nbRemovals ← revise(C, Y)
8:       if nbRemovals > 0 then
9:          if dom(Y) = ∅ then return FAILURE
10:         for each C' | C' ≠ C ∧ Y ∈ vars(C') do
11:            Q ← Q ∪ {C'}
12:            ctr(C', Y) ← ctr(C', Y) + nbRemovals
13:         end for
14:      end if
15:   end for
16:   for each Y ∈ vars(C) do ctr(C, Y) ← 0
17: end while
18: return SUCCESS
```

then there is no need to perform again the revision of $(C_{i,j}, X_j)$ if X_i is again selected and if the domain of X_i has not been modified elsewhere. As another illustration (as expressed in [5]), let us assume a ternary constraint $C_{i,j,k}$. If the selection of the variable X_i involves a revision of $(C_{i,j,k}, X_j)$ and of $(C_{i,j,k}, X_k)$ then there is no need to perform again the revision of $(C_{i,j,k}, X_k)$ if the variable X_j is selected and if the domains of X_i and of X_j have not been modified elsewhere.

By associating a counter $ctr(C, X)$ with any arc, it is possible to determine which revisions are relevant. The value of $ctr(C, X)$ denotes the number of removed values in $dom(X)$ since the last revision involving C. Initially, this value is arbitrarily fixed to 1 for all counters. Then, when a variable X is selected and when a constraint C binding X is considered, two situations can happen. If X is the only variable in $vars(C)$ such that $ctr(C, X) > 0$, then the revision of all arcs (C, Y) with $Y \neq X$ is performed. Otherwise (second situation), all arcs (C, Y), including $Y = X$, are revised. Indeed, it is also relevant to revise (C, X) since at least another variable of C has been modified elsewhere. This is the function *needsNotBeRevised*, described by Algorithm 4 which allows determining whether the revision of an arc is relevant. When taking into consideration the second situation and the fact that all counters related to C are reinitialized to 0 after C has been considered, the test of the line 8 of Algorithm 3 becomes meaningful: it allows avoiding useless revisions.

3.3 Constraint-oriented AC3

The third AC3 variant uses a constraint-oriented propagation scheme (as $AC3_d$ can be regarded in the binary case) and is depicted in Algorithm 5. The

principle is to insert in Q all constraints for which at least a revision is necessary. Initially, all constraints are inserted in Q. Then, iteratively, each constraint C of Q is selected and each variable X of $vars(C)$ is considered. Similarly as the variable-oriented variant, the introduction of some counters allows avoiding useless revisions.

4 Revision ordering heuristics

At this step, it is natural to wonder about the practical interest of the variable-oriented and constraint-oriented variants. Indeed, as the three variants seem to be equivalent, we could have just introduce the arc-oriented one since it is simpler and more natural. The answer is that the nature of the elements of the set Q can have important repercussions on the overall behaviour of the algorithms. From a certain perspective, the variable-oriented and constraint-oriented variants have a grain bigger than the arc-oriented one. Instead of being a drawback (as it seems to be at first sight), it can be in fact an advantage. This is what we are going to show by introducing so-called revision ordering heuristics, i.e., heuristics to order the revisions to be applied by AC3 (or its extensions)[1].

Some of the heuristics that we introduce here are original and some other are simply taken from or adaptations of previous works (a discussion about related work is proposed later in this section). But, first, we present the revision ordering heuristic $fifo$ that can be defined without any ambiguity whatever variant is chosen. This heuristic involves selecting the oldest element of Q (viewed as a queue).

On the other hand, from now on, we shall introduce the composition of heuristics whose operator is denoted $\circ$ (as in [19]). A composed heuristic $h_2 \circ h_1$ means that the heuristic h_1 is used first, and then, if necessary, the heuristic h_2 is used to break ties (a tie is a set of elements that are considered as equivalent by an heuristic). For all heuristics (including composed ones) presented below, when a tie is still to be broken, the oldest element of the tie is selected (using implicitly $fifo$). Note that other "final" tie-breakers could be studied.

4.1 Variable-oriented heuristics

Each of the variable-oriented heuristics, i.e., heuristics adapted to the variable-oriented variant, selects a variable X from Q with:

- dom^v: the smallest current domain size,
- rem^v: the greatest proportion of removed values in its domain.
- $ddeg$: the greatest current (also called dynamic) degree,

In some way, dom^v and rem^v are complementary. The former is based on the number of remaining values whereas the latter is based on the number (proportion) of removed values (since the last selection of the variable). Note that

[1] We think that the term of "revision ordering heuristics" is more appropriate than "ordering heuristics" [21], "constraint ordering heuristics" [8] or "arc heuristics" [18].

considering a raw number of removed values is in favour of large domains (since there are more opportunities to remove values) and, consequently, usually entails more constraint checks. This is the reason why we have preferred using proportions.

4.2 Constraint-oriented heuristics

Each of the constraint-oriented heuristics, i.e. heuristics adapted to the constraint-oriented variant, selects a constraint C from Q with:

- dom^c: the smallest current domain size,
- rem^c: the greatest proportion of removed values in its domain.

The heuristics dom^c and rem^c are similar to dom^v and rem^v. Remember that we call current domain of a constraint C the Cartesian product of the current domains associated with the variables in $vars(C)$. It must not be confused with the constraint size or satisfiability of [21]. Hence, the removed values from the domain of a constraint correspond to the removed tuples due to the modification of the domains of some variables. For instance, let us consider a binary constraint $C_{i,j}$ involving two variables X_i and X_j. Assume that, at last selection of $C_{i,j}$, the domains of X_i and X_j had 10 values. Then, if, at current selection, the domains respectively have 6 and 8 values, the size of the current domain $C_{i,j}$ is 48 and the proportion of removed values is 52/100.

4.3 Arc-oriented heuristics

Each of the arc-oriented heuristics, i.e., heuristics adapted to the arc-oriented variant, selects an arc (C, X) from Q with:

- dom^v: the variable which has the smallest current domain size,
- dom^c/dom^v: the smallest ratio between the current domain size of the constraint (i.e., the number of tuples in the Cartesian product built from the current domains attached to the variables involved in the constraint) and the current domain size of the variable,
- $ddeg \circ dom^v$: the variable which has the smallest current domain size, and in case of equivalence, the greatest current degree.

4.4 Related work

In this subsection, we present some works related to revision ordering heuristics. First, let us cite the seminal work of [21] which propose different revision ordering heuristics to be used with the arc-oriented variant of AC3. These heuristics devised for binary problems are based on three major features:

- the number of supports in each constraint (called satisfiability),
- the number of values in the domain of each variable,
- the degree of each variable.

The heuristic *sat up* is based on satisfiability and allows to obtain interesting results in terms of constraint checks. However, determining the number of supports in each constraint and maintaining this information is not a very practical approach. This observation has been stated by [8] which propose another heuristic κ_{ac} based on the number of supports in each constraint in order to minimizing the constrainedness of the resulting subproblem. Some experiments [8] show that κ_{ac}, which is time expensive, performs less constraint checks than *sat up* and *dom j up* at the phase transition of arc consistency.

The heuristic *dom j up* selects the arc $(C_{i,j}, X_i)$ such that the variable X_j, i.e. the variable relaxed against, has the smallest current domain size. With respect to our notation, this heuristic corresponds to dom^c/dom^v (when considering binary problems). There is also a correspondence between *dom j up* and the heuristic dom^v defined for the variable-oriented variant.

The heuristic *deg down* which corresponds to *ddeg* is particularly disappointing in the experiments of [21]. We have made the same observation.

On the other hand, [15] mentions the issue of ordering the removed values which are put in different queues (a queue per variable). However, this approach is specific to fine-grained algorithms. [10] propose an original approach by managing propagation events associated with variables. Each event entails the immediate propagation of some constraints binding the variable associated with this event. A layered propagation architecture schedules the propagation of constraints according to a compromise between the provided information and the computation cost.

Finally, let us mention the work of [18] which emphasizes the importance of revision ordering heuristics as well as so-called domain heuristics. In [19], a precise revision ordering heuristic, called *comp* is presented as being tuned for $AC3_d$ (an arc-oriented AC3-based variant). It is a heuristic composed of 6 basic criteria. Roughly speaking, it corresponds to (when only considering the two first criteria) $ddeg \circ dom^v$.

5 Experiments

To compare the efficiency of the heuristics introduced in this paper, we have implemented them in Java and performed some experiments (run on a PC Pentium IV 2,4GHz 512MB under Linux) with respect to random, academic and real-world problems. Performances have been measured in terms of the CPU time in seconds (time), the number of constraint checks (#ccks) and the number of times (#rohs) a revision ordering heuristic has to select an element in the propagation set Q. The arc consistency algorithm that has been used for our experimentation is AC3.2 [11].

5.1 Stand-alone arc consistency

First, we have considered stand alone arc consistency which involves making arc consistent a CSP instance (that is to say, no search is performed). The first

		P1			P2		
variant	heuristic	time	#ccks	#rohs	time	#ccks	#rohs
variable	$fifo$	0.064	94,012	150	0.130	326,752	67
variable	dom^v	0.065	94,012	150	0.030	63,540	55
variable	rem^v	0.065	94,012	150	0.037	85,152	26
variable	$ddeg$	0.066	94,012	150	0.049	102,379	36
arc	$fifo$	0.071	94,012	1000	0.177	441,859	927
arc	dom^v	0.094	94,012	1000	0.120	204,346	895
arc	dom^c/dom^v	0.195	94,012	1000	0.297	113,798	1,060
arc	$ddeg \circ dom^v$	0.151	94,014	1000	0.119	115,277	490
constraint	$fifo$	0.063	94,012	500	0.188	484,108	600
constraint	dom^c	0.096	94,012	500	0.079	42,884	463
constraint	rem^c	0.103	94,012	500	0.053	77,299	131

Table 1. Stand alone arc consistency on random instances

series of experiments that we have run corresponds to some random problems. In this paper, a class of random CSP instances will be characterized by a 4-tuple $<n, d, m, t>$ where n is the number of variables, d the uniform domain size, m the number of binary constraints and t the number of unallowed tuples.

We present the results, given in Table 1 and Table 2, about some random binary instances studied in [3, 5, 22]. More precisely, 4 classes, denoted here P1, P2, P3 and P4, have been experimented. P1 = $<150, 50, 500, 1250>$ and P2 = $<150, 50, 500, 2350>$ respectively correspond to classes of under-constrained and over-constrained instances. P3=$<150, 50, 500, 2296>$ and P4=$<50, 50, 1225, 2188>$ correspond to classes of instances at the phase transition of arc consistency for sparse problems and for dense problems, respectively. For each class, mean results are given for 50 generated instances using the generator of [7].

		P3			P4		
variant	heuristic	time	#ccks	#rohs	time	#ccks	#rohs
variable	$fifo$	0,244	546,900	701	0.446	911,748	203
variable	dom^v	0.219	478,135	708	0.426	857,789	225
variable	rem^v	0.233	519,207	525	0.433	888,890	173
variable	$ddeg$	0.253	500,287	669	0.473	911,748	203
arc	$fifo$	0.266	569,228	6,068	0.517	944,679	12,659
arc	dom^v	0.333	518,310	6,301	1.034	867,232	15,195
arc	dom^c/dom^v	0.903	506,318	7,456	4.423	882,329	16,926
arc	$ddeg \circ dom^v$	0.490	493,795	5,781	1.706	867,232	15,195
constraint	$fifo$	0.252	561,398	3,740	0.460	927,966	7,571
constraint	dom^c	0.539	459,739	5,652	2.098	823,478	13,051
constraint	rem^c	0.496	527,545	2,740	2.095	897,306	6,055

Table 2. Stand alone arc consistency on random instances

One can immediately notice that the number of heuristic solicitations (#rohs) is far less important for the variable-oriented variant. In fact, when a variable is

variant	heuristic	SCEN#05 time	SCEN#05 #ccks	SCEN#05 #rohs	SCEN#08 time	SCEN#08 #ccks	SCEN#08 #rohs
variable	$fifo$	0.276	899,793	1,184	0.243	830,824	212
variable	dom^v	0.159	294,882	625	0.095	39,795	69
variable	rem^v	0.228	585,194	815	0.119	150,047	64
variable	$ddeg$	0.286	652,228	943	0.214	365,830	125
arc	$fifo$	0.315	981,555	19,701	0.560	2,178,674	17,040
arc	dom^v	2.291	742,330	20,731	4.478	876,989	8,067
arc	dom^c/dom^v	7.921	251,064	8,567	4.662	30,674	816
arc	$ddeg \circ dom^v$	4.916	687,107	18,660	10.099	904,712	8,748
constraint	$fifo$	0.303	964,764	12,000	0.461	1,877,001	5,508
constraint	dom^c	4.277	261,892	9,010	2.043	25,028	781
constraint	rem^c	4.890	298,310	7,036	3.112	74,614	984

Table 3. Stand alone arc consistency on RLFAP instances

selected in the propagation set Q, it entails a number of revisions related to the degree of the variable. On the other hand, for the arc-oriented variant, when an arc is selected, it just entails one revision, and, for the constraint-oriented variant, when a k-ary constraint is selected, it entails at most k revisions. Hence, the variable-oriented variant has a bigger grain than the other variants: the number of constraints checks performed after each solicitation is far more important.

We also observe that the variable-oriented variant clearly appears to be the fastest one when using some revision ordering heuristics, and, more precisely, the heuristic dom^v. This behaviour can be explained as follows. The variable-oriented variant requires less solicitations, as stated above, and each solicitation is cheap. Indeed, the overhead of picking the best element in the propagation set is limited for the variable-oriented variant, unlike other ones, since there are less variables than constraints and arcs.

In terms of constraint checks, the best heuristics are the constraint-oriented heuristic dom^c and the variable-oriented dom^v whereas the worse heuristics are the three versions of $fifo$. However, the time performance of these "standard" heuristics is not too bad as, systematically, the first element of the propagation set is selected.

Next, we have tested real-world instances, taken from the FullRLFAP archive[2], which contains instances of radio link frequency assignment problems. Table 3 presents the results obtained for two instances, denoted SCEN#05 and SCEN#08, studied in [3, 18, 22], and confirms all remarks expressed above. Note that there is a gap between the standard heuristics $fifo$ and some other heuristics with respect to the number of constraints checks required for SCEN#08. A similar behaviour has been observed by [18].

5.2 Maintaining arc consistency during search

As it appears that one of the most efficient complete search algorithms is the algorithm which Maintains Arc Consistency during the search of a solution [16,

[2] We thank the Centre d'Electronique de l'Armement (France).

variant	heuristic	bqwh-15-106		Q1		Q2	
		time	#ccks	time	#ccks	time	#ccks
variable	$fifo$	4.68	1.601M	166.43	100.192M	44.21	15.321M
variable	dom^v	3.87	1.211M	122.77	66.270M	39.86	8.860M
variable	rem^v	4.11	1.270M	132.22	77.955M	45.59	11.292M
variable	$ddeg$	5.57	1.879M	194.13	102.730M	71.76	14.565M
arc	$fifo$	4.76	1.615M	173.26	103.099M	44.20	15.639M
arc	dom^v	6.28	0.921M	251.91	60.276M	69.39	9.329M
arc	dom^c/dom^v	26.81	1.193M	932.59	66.183M	124.54	8.858M
arc	$ddeg \circ dom^v$	28.91	1.015M	1,170.90	60.305M	222.01	9.317M
constraint	$fifo$	4.66	1.599M	169.65	100.915M	43.24	15.444M
constraint	dom^c	11.41	0.858M	491.94	52.753M	94.46	7.060M
constraint	rem^c	19.28	1.306M	809.09	80.343M	144.21	11.528M

Table 4. Maintaining arc consistency on random instances

4], we have implemented a MAC3.2 version which integrates the $dom/ddeg$ [4, 17] variable ordering heuristic, and the lexicographic value ordering heuristic.

First, to study the behaviour of the different heuristics wrt problems involving random generation, we have considered the following classes of instances :

- one class, denoted bqwh-15-106, of 100 satisfiable balanced Quasigroup With Holes (bQWH) instances [9] of order 15 with 106 holes,
- two classes Q1=<80, 10, 400, 35> and Q2=<900, 10, 1250, 70> of 100 random binary instances situated at the phase transition of search for relatively dense problems ($\approx 12\%$) with low tightness (35%) and sparse problems ($\approx 0.3\%$) with high tightness ($\approx 70\%$), respectively.

Table 4 presents the results obtained for all these classes. When considering the number of constraint checks, one can observe that all heuristics based on dom^c or dom^v are the best ones. More precisely, the constraint-oriented heuristic dom^c outperforms all other ones and saves about 50% of the constraint checks required by $fifo$ heuristics. We also note that the coarser grain of the variable-

variant	heuristic	SCEN#11			GRAPH#14		
		time	#ccks	#rohs	time	#ccks	#rohs
variable	$fifo$	88.950	36.379M	470,966	1.713	1.237M	5,700
variable	dom^v	80.547	22.238M	308,608	1.730	1.189M	4,428
variable	rem^v	82.461	22.329M	310,253	1.775	1.188M	4,428
variable	$ddeg$	98.475	34.808M	595,373	2.666	1.215M	4,978
arc	$fifo$	90.381	33.688M	7,548,869	1.711	1.237M	39,838
arc	dom^v	93.596	18.511M	4,781,551	4.271	1.218M	35,870
arc	dom^c/dom^v	214.370	21.169M	4,336,842	22.552	1.180M	30,517
arc	$ddeg \circ dom^v$	410.737	18.775M	4,833,847	24.614	1.218M	35,961
constraint	$fifo$	90.054	36.130M	5,586,648	1.708	1.237M	30,030
constraint	dom^c	111.995	17.318M	3,256,374	6.901	1.190M	24,032
constraint	rem^c	158.468	22.079M	4,136,456	7.847	1.188M	23,197

Table 5. Maintaining arc consistency on RLFAP instances

variant	heuristic	*cc*-7-2	*cc*-7-3	*gr*-34-9	*gr*-34-10	*qa*-5	*qa*-6
variable	$fifo$	9.159	238.630	47.853	1,736.913	62.104	5,943.426
variable	dom^v	8.688	213.966	31.238	1,174.526	56.138	4,331.342
variable	rem^v	8.531	222.600	35.666	1,292.841	57.310	4,636.216
variable	$ddeg$	8.955	291.663	52.885	1,907.120	69.987	6,435.984
arc	$fifo$	10.263	276.240	56.017	1,997.052	68.729	5,679.748
arc	dom^v	39.856	735.510	90.419	5,025.990	129.377	10,096.466
arc	dom^c/dom^v	129.977	2,636.612	251.786	13,871.150	457.250	49,364.050
arc	$ddeg \circ dom^v$	498.214	7,918.398	1,121.336	109,292.280	719.742	126,063.110
constraint	$fifo$	9.342	229.128	52.504	1,951.002	65.983	5,516.274
constraint	dom^c	19.926	736.137	98.216	6,747.980	205.111	18,382.519
constraint	rem^c	38.754	1,210.565	528.161	30,428.539	335.617	33,818.809

Table 6. cpu time when maintaining arc consistency on academic instances

variant	heuristic	*cc*-7-2	*cc*-7-3	*gr*-34-9	*gr*-34-10	*qa*-5	*qa*-6
variable	$fifo$	3.196M	116.585M	42.836M	1,609.906M	56.326M	3,584.318M
variable	dom^v	3.196M	111.633M	34.882M	1,347.494M	47.610M	2,765.204M
variable	rem^v	3.196M	111.652M	38.052M	1,434.013M	48.741M	2,876.704M
variable	$ddeg$	2.977M	121.794M	46.012M	1,660.198M	56.327M	3,584.320M
arc	$fifo$	3.155M	118.312M	45.175M	1,680.035M	58.098M	3,631.175M
arc	dom^v	2.261M	93.980M	26.979M	1,028.886M	42.862M	2,179.902M
arc	dom^c/dom^v	1.766M	92.250M	32.315M	1,213.946M	45.920M	2,677.797M
arc	$ddeg \circ dom^v$	2.503M	106.447M	27.121M	1,031.489M	42.861M	2,179.929M
constraint	$fifo$	3.196M	116.252M	44.733M	1,684.691M	56.366M	3,580.447M
constraint	dom^c	1.366M	80.645M	29.084M	1,115.535M	40.467M	2,125.472M
constraint	rem^c	2.843M	113.639M	40.145M	1,462.286M	48.367M	2,882.489M

Table 7. #ccks when maintaining arc consistency on academic instances

oriented heuristics has an impact on the number of constraint checks (dom^v entails more checks than dom^c). However, it is highly compensated by the small overhead of such heuristics as explained above.

Finally, we introduce three additional tables that confirm our previous results. Table 5 corresponds to the real-world instances SCEN#11 and GRAPH #14 of the RLFAP archive. Tables 6 and 7 respectively correspond to the cpu time and the number of constraint checks required to solve the following academic instances:

- two chessboard coloring instances [1], denoted *cc*-7-2 and *cc*-7-3, involving quaternary constraints,
- two Golomb ruler instances[3], denoted *gr*-44-9 and *gr*-44-10, involving binary and ternary constraints,
- two prime queen attacking instances[4], denoted *qa*-5 and *qa*-6, involving only binary constraints.

To summarize, the efficiency of all variable-oriented heuristics based on current domain sizes has been established both for stand alone arc consistency and

[3] See problem006 at `http://4c.ucc.ie/~tw/csplib/`
[4] See problem029 at `http://4c.ucc.ie/~tw/csplib/`

for MAC. On the other hand, the arc-oriented and constraint-oriented heuristics which allow saving some constraint checks, are quite costly. Nevertheless, there are at least three alternatives to improve all heuristics:

- restricting the search of the best element to a subset of Q,
- performing the search of the k best elements every k selections,
- improving the management of Q using a pigeonhole sort [21].

Such an optimization is proposed by [19] with the heuristic *comp* which belongs to the group of the best heuristics (when considering constraint checks). [19] uses an efficient representation for the queue which allows compensating the time spent on selection and maintenance.

6 Conclusion

Our first motivation, in this paper, was to clarify the situation about the different propagation schemes of AC3-based algorithms. Indeed, we can define three variants which respectively correspond to algorithms with an arc-oriented, variable-oriented and constraint-oriented propagation scheme. We have presented general versions of these algorithms so that they can be applied to non-binary problems whereas being careful about avoiding useless revisions (for variable-oriented and constraint-oriented variants). To determine which variant is the more appropriate to establish arc-consistency, we have studied the impact of introducing so-called revision ordering heuristics. Such heuristics have been proposed by [21] with respect to the arc-oriented variant and experimentations performed when arc consistency is used as a preprocessing of binary problems.

In this paper, we have extended our understanding of revision ordering heuristics by:

- introducing new heuristics and adapting heuristics of [21] with respect to the different variants of AC3-based algorithms,
- experimenting these heuristics when arc consistency is maintained during the search of a solution for binary and non binary problems.

Experimental results show that heuristics based on the number of removed values are disappointing, unlike heuristics based on the number of remaining values. The best heuristics save up to 50% of constraint checks when compared to the "standard" heuristic $fifo$. Hence, it confirms for MAC the observation of [21]. Also, the best variable-oriented heuristics can save about 25% of cpu-time when compared to $fifo$. However, it turns out that constraint-oriented and arc-oriented heuristics are penalized in terms of CPU time. The reason is that the constraint-oriented and arc-oriented variants need to record more elements in the propagation set Q than the variable-oriented variant. Hence, both variants are time-consuming when one heuristic iterates all recorded elements in Q.

To conclude, even if there exists some perspectives to optimize all these heuristics by avoiding systematic iterations of Q, we believe that a AC3-based variable-oriented variant associated with a revision ordering heuristic based on dom^v should preserve its advantage (as it could also benefit from such improvements).

References

1. M. Beresin, E. Levin, and J. Winn. A chessboard coloring problem. *The College Mathematics Journal*, 20(2):106–114, 1989.
2. C. Bessière. Arc consistency and arc consistency again. *Artificial Intelligence*, 65:179–190, 1994.
3. C. Bessière, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.
4. C. Bessière and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
5. C. Bessière and J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, 2001.
6. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.
7. D. Frost, R. Dechter, C. Bessière, and J.C. Régin. Random uniform CSP generators. http://www.lirmm.fr/~bessiere/generator.html, 1996.
8. I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constraindedness of arc consistency. In *Proceedings of CP'97*, pages 327–340, 1997.
9. C.P. Gomez and D. Shmoys. Completing quasigroups or latin squares: a structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalization*, 2002.
10. F. Laburthe and le projet OCRE. CHOCO : implémentation du noyau d'un système de contraintes. In *Actes de JNPC'00*, pages 151–165, 2000.
11. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.
12. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:118–126, 1977.
13. J.J. McGregor. Relational consistency algorithms and their applications in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.
14. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
15. J.C. Régin. *Développement d'outils algorithmiques pour l'intelligance artificielle. Application à la chimie organique*. PhD thesis, Universit Montpellier II, 1995.
16. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the PPCPA'94*, Seattle WA, 1994.
17. B.M. Smith and S.A. Grant. Trying harder to fail first. In *Proceedings of ECAI'98*, pages 249–253, Brighton, UK, 1998.
18. M.R.C. van Dongen. $AC3_d$ an efficient arc consistency algorithm with a low space complexity. In *Proceedings of CP'02*, pages 755–760, 2002.
19. M.R.C. van Dongen. Lightweight arc-consistency algorithms. Technical Report TR-01-2003, University college Cork, 2003.
20. R.J. Wallace. Why AC3 is almost always better than AC4 for establishing arc consistency in CSPs. In *Proceedings of IJCAI'93*, pages 239–245, 1993.
21. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of NCCAI'92*, pages 163–169, 1992.
22. Y. Zhang and R.H.C. Yap. Making AC3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle WA, 2001.

New Light on Arc-Consistency over Continuous Domains

Gilles Chabert, Gilles Trombettoni, and Bertrand Neveu

PROJET COPRIN I3S-INRIA-CERTIS, 2004 route des Lucioles BP 93
06902 Sophia Antipolis Cedex, FRANCE
{chabert,trombe,neveu}@sophia.inria.fr

Abstract. Hyvönen [5] and Faltings [4] observed that propagation algorithms with continuous variables are computationally extremely inefficient when unions of intervals are used to precisely store refinements of domains.
These algorithms were designed in the hope of obtaining the interesting property of arc-consistency, that guarantees every value in domains to be consistent w.r.t. every constraint.
In this paper, we show that a pure backtrack-free filtering algorithm enforcing arc-consistency will never exist. But surprisingly, we show that it is easy to obtain a property stronger than arc-consistency with a few steps of bisection.
We define this so-called *box-set consistency* and detail an efficient algorithm to enforce it.

1 Introduction

Solving systems of nonlinear equations over the reals with interval constraint programming usually resorts to a combination of local filtering, interval analysis and domain splitting.

Local filtering techniques are based on an interval narrowing operator, called *projection*, that computes compatible values for different variables linked by a constraint. Sometimes, the projection results in a union of intervals: for instance with the constraint $x^2 = y$, if y varies within $[1, 4]$ then the domain of variation for x obtained by projection is either $[-2, -1]$ or $[\ 1,\ 2]$.

In this case, the *hull consistency* algorithm [1] (also known as *2B consistency* [6]) computes the enclosing interval of the union immediately, therefore losing track of each "gap" between intervals. Another approach, inspired by the well-known *arc-consistency*, would be to store and propagate unions of intervals.

Arc-consistency has never been applied successfully. We explain with an example that this failure is not due to bad algorithmic choices, but to a property inherent to continuous CSPs.

Yet, we show that it is possible in a solving strategy that includes a specific splitting technique, called *natural splitting*, to obtain boxes that verify not only arc consistency but another property called *box-continuity*. The lazy version of this new algorithm causes no time overhead in practice and gives promises.

The paper is organized as follows. In section 2, we sum up the concepts of constraint programming over the reals. In section 3, we explain why arc-consistency cannot be achieved. We define a stronger consistency and show related properties. In section 4, an algorithm that enforces this consistency is given.

2 Background

2.1 Constraint Reasoning over the Reals

A **numerical constraint satisfaction problem** (NCSP) is a 3-uple (C,V,B). C is a set of constraints $c_1, ..., c_m$ (equations or inequations) relating a set V of variables $x_1, ..., x_n$. Each variable is given an initial domain of real values $D_{x_1}, .., D_{x_n}$, and the problem is to find all the n-tuples of values $(v_1, ..., v_n)$, $v_i \in D_{x_i}$ $(1 \leq i \leq n)$, such that constraints are all satisfied when simultaneously each variable x_i is assigned to v_i. Such a n-tuple is called a *solution*. Usually, domains are represented by intervals and a *box* designates a cartesian product of domains. $B = D_{x_1} \times ... \times D_{x_n}$ is the initial box of the problem. In this paper, we will resort also to a more complex representation of domains, where a variable domain is assigned a union of intervals. In this case, the cartesian product B of domains will be called a *h-box* (a box with "gaps").

We will use intensively a relation of problem inclusion. Let us define it once and for all.

Definition 1 (Sub-NCSP). *Let $P = (C, V, B)$ and $P' = (C', V', B')$ be two NCSP. P is included in P' iff $C = C'$, $V = V'$ and $B \subset B'$*

In practice, NCSP are large nonlinear problems that are intractable by symbolic solving techniques. Traditional numerical methods do not suit either because we are looking for all the solutions. Solving can be achieved by combining local filtering, domain splitting, and interval analysis.

Local filtering techniques refine domains of variables thanks to partial properties of the problem, that is, properties which hold on subproblems. These techniques converge in polynomial time, and the resulting box (or h-box) is said to be *locally* consistent.

The general scheme for finding solutions consists in a search tree, where local filtering is enforced at each node. Once local consistency is reached, the domain of one variable is chosen and split in two sub-domains, which leads to two subnodes in the tree.

A major approach for local filtering is the well-known *hull consistency*, obtained by a Waltz-like propagation algorithm [9]. The algorithm is detailed further. Here are the underlying concepts :

- **Projection**: Refine the domain of a variable x with respect to a specific constraint c, using interval arithmetics [8].
- **Propagation**: Propagate reductions over the other variables linked to x by another constraint c'.

2.2 Projections

Let c be a binary constraint relating variables x and y. We denote Π_x^c the *projection* [1] *of* c *over* x, a function that takes an h-box $B = D_{x_1} \times ... \times D_x \times ... \times D_y \times ... \times D_{x_n}$ as input and computes all possible values for x in D_x as y varies within D_y. Formally, if D'_x is the result of Π_x^c applied on B, we have: $D'_x = \{v \in D_x \mid \exists w \in D_y, \quad c(v, w) \text{ is satisfied}\}$. This definition can be easily generalized to k-ary constraints:

Definition 2 (Projection). *Let* c *be a constraint relating variables* $x, y_1, ..., y_k$. *We call projection of* c *over* x *the following function:*
$\Pi_x^c : B \rightarrow \{v \in D_x \mid \exists (v_1, ..., v_k) \in D_{y_1} \times ... \times D_{y_k}, \quad c(v, v_1, ..., v_k) \text{ is satisfied}\}$

Example 1. $c : x + y = z$
$\Pi_x^c : D_x \times D_y \times D_z \longrightarrow (D_z \ominus D_y) \cap D_x$
where $\ominus$ is the natural extension of the arithmetic operator minus. Note that computing this projection requires also an intersection with D_x.

Basically, the projection of a constraint $f(y, x_1, ..., x_n) = 0$ over y requires to find an implicit function ϕ such that $f(y, x_1, ..., x_n) = 0 \Leftrightarrow y = \phi(x_1, ..., x_n)$.

Sometimes, there is not a unique implicit function but several continuous functions $(\phi_1, \phi_2, ...)$, then we talk about *disjunction* and in this case Π_y^c gives a union of intervals:

Example 2. $c : x^2 = y$
Π_x^c applied on $D_x \times D_y$ with $D_x = [-2, 2]$ and $D_y = [1, 4]$ gives the union $D'_x = [-2, -1] \cup [\ 1,\ 2]$

The set of values returned by a projection may be either an interval (example 1) or a union of intervals (example 2). To place our discussion in the most general case, we consider hereinafter that a projection returns a set U of disjoint intervals, that we will call abusively a *union*, and write $|U|$ the number of intervals contained in U [2].

2.3 Propagation

Modifying (or revising) the domain of a variable may have repercussions on the other variables. *Propagate* means to memorize in a queue (or an agenda) all the pairs $< c, x >$ of constraint/variable such that the projection of c over x can be effective. When the queue is empty, we are sure that no more reduction is possible and that we have reached a fix point.

[1] Also called *Solution function* [5].

[2] Unions of disjoint intervals could be defined algebraically with their operators and their arithmetic. But their use is rather intuitive, so we will not give such a formalism here. Sometimes we just substitute unions for intervals, to avoid to overwhelm this paper with definitions.

In this paper, we will refer to a procedure `Propagate(NCSP` (C, V, B)`, in-out Queue` Q`, Constraint` c`, Variable` x`)`, that updates the propagation queue Q after a projection of c over x.

If revising a domain consists in applying the projection operator Π defined above, the resulting NCSP is *arc-consistent*:

Definition 3 (Arc-Consistency). *Let* $P = (C, V, B = D_{x_1} \times ... \times D_{x_n})$ *be a NCSP.*
P is arc consistent iff $\forall < c, x > \in C \times V$ *with x related by c,* $D_x = \Pi_x^c(B)$

Remark 1. In this paper, we will talk about the *arc-consistency of a box (or an h-box)* B to designate the arc-consistency of the problem (C, V, B) with the set C and V given by the context.

We will see in section 3.1 that arc-consistency is not feasible with continuous variables, so usually the revising operation is not the projection itself, but an outer approximation. Computations are all interval-based and this operator avoids to manage unions. The resulting problem is *hull-consistent*:

Definition 4 (Hull-Consistency). *Let* $P = (C, V, B = D_{x_1} \times ... \times D_{x_n})$ *be a NCSP.*
P is hull consistent iff $\forall < c, x > \in C \times V$ *with x related by c,* $D_x = \square\Pi_x^c(B)$

The symbol $\square$ stands for the *hull* operation. Example: $\square\{[0,1],[2,3]\} = [0,3]$.

Propagation, arc-consistency, and hull consistency are extensively covered in literature, see [2] for example. To summarize, here is a generic algorithm `HC_Filtering` of hull consistency filtering.

Procedure 1 `HC_Filtering`(NCSP (C, V, B))

```
var Q : Queue
for all pairs < c, x > in C × V do
  if x is related by c then
    add < c, x > in Q
while Q is not empty do
  pop a pair < c, x > from Q
  D'_x ⟵ □ Π^c_x(B)
  if D'_x ⊂ D_x then
    Propagate((C, V, B), Q, c, x)
    D_x ← D'_x                    // D_x is the domain of x in B
```

This algorithm originates from Waltz [9] and was applied first over finite domain constraints under the acronym `AC3` [7] to obtain arc-consistency. With intervals, `HC3` [3] introduces a decomposition of the system into *primitive constraints* [3] for which projections can be computed, and `HC4` [2] is an upgraded

[3] A *primitive constraint* is a basic mathematical relation (such as $z = x + y$ or $y = cos(x)$) for which projections are known. A system of standard equations can always be decomposed into an equivalent system of primitive constraints.

version of HC3 that produces the same result sparing decomposition. Both enforce hull-consistency.

3 A stronger property than arc-consistency

In this section, we introduce a new kind of consistency, on a theoretical point of view. The rest of the paper will be devoted to the way it can be enforced. Contrary to arc-consistency, it can be obtained in reasonable time and even more, in some cases, provide improvements compared to the classical approach just given above.

3.1 Arc-Consistency

First of all, let us rule out an ambiguity. Talking about the *arc-consistency of a problem* may have two different meanings, depending on the context. We may refer to the property, which can be either true or false. But we may talk also about the *largest arc-consistent subproblem*. In the latter case, we will use the following definition:

Definition 5 (AC Part). *The AC part of a NCSP is the maximal arc-consistent sub-NCSP, for the order relation of inclusion (see definition 1).*

Example 3. Let $P = (\{x = y\}, \{x, y\}, D_x \times D_y)$ be a NCSP with $D_x = [-1, 1]$ and $D_y = [0, 2]$

In the AC part of P, domains become $[0, 1] \times [0, 1]$. Indeed, any arc-consistent sub-NCSP of P has an h-box with intervals $[\alpha, \beta]$, $0 \leq \alpha \leq \beta \leq 1$, and the (unique) maximal element of these h-boxes is $[0, 1] \times [0, 1]$.

We show below that even with very simple constraints, the AC part of a problem may have a non-representable domain, as an infinity of intervals. Hence, arc-consistency filtering is not applicable over continuous domains, whatever the underlying algorithm is.

We are going to illustrate our claim on the following system of 2 equations:

Example 4. Let $P = (\{c_1, c_2\}, \{x, y\}, B)$ be the following $NCSP$:
$B = D_x \times D_y = [1, 9] \times [1, 9]$
(c_1) : $(\frac{3}{4}(x - 5))^2 = y$
(c_2) : $y = x$

Lemma 1. *In the AC part of P, domains of x and y are an infinity of disjoint non-empty intervals.*

Necessary condition Let f_1 and f_2 be the following (real-valued) functions:
$f_1 : y \longrightarrow \frac{4}{3}\sqrt{y} + 5$
$f_2 : y \longrightarrow 5 - \frac{4}{3}\sqrt{y}$
Let F_1 (resp. F_2) be the "optimal" extensions to intervals [4] of f_1 (resp. f_2), Φ_1 (resp. Φ_2) the extension to unions of intervals associated to F_1 (resp. F_2). Finally, let Φ be the function such that $\Phi(U) = \Phi_1(U) \cup \Phi_2(U)$.

Consider an algorithm that, in turn, computes the following operations: $D_x \leftarrow \Phi(D_y)$ et $D_y \leftarrow D_x$. We omit intersections with domains on purpose, and this is why, *a priori*, we do not call these operations *projections* . Let us denote X_n (resp. Y_n) the domain of x (resp. y) after the n^{th} execution of $D_x \leftarrow \Phi(D_y)$ (resp. $D_y \leftarrow D_x$). X_0 and Y_0 are initial domains.

The figures below depict the first steps of propagation. The h-boxes shown are successively $X_0 \times Y_0$, $X_1 \times Y_0$, $X_1 \times Y_1$ and $X_2 \times Y_2$.

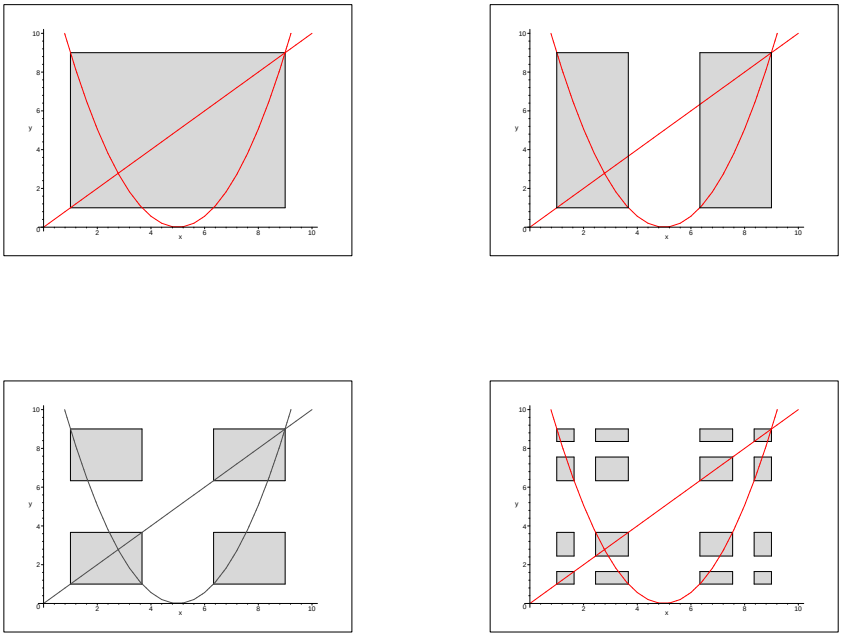


Fig. 1. First steps of `AC` filtering

As we see, the size of unions grows exponentially. Let us show some properties of this algorithm.

[4] F is optimal iff for any interval I, $F(I) = \Box f(I)$

Property 1. $\forall n \geq 1,\ \ X_n \subset X_{n-1}$ and $Y_n \subset Y_{n-1}$. In other words, the result of each operation is included in the current domain of the variable.

Proof. By induction. We can check by hand that $X_1 \subset X_0$ and $Y_1 \subset Y_0$. Assume $X_n \subset X_{n-1}$:
$X_n \subset X_{n-1} \Longrightarrow Y_{n+1} \subset Y_n \Longrightarrow \Phi(Y_{n+1}) \subset \Phi(Y_n)$ because interval arithmetic is inclusion monotonic, and then $X_{n+1} \subset X_n$. ▲

Property 2. The number of intervals doubles at each step ($\forall n\ |X_n| = 2\times|X_{n-1}|$), and more precisely, each interval is split into two disjoint intervals.

Proof. Assume that X_n and Y_n contain p disjoint intervals whose bounds are between 1 and 9. As functions f_1 and f_2 are monotonous on $[1,9]$, $\Phi_1(Y_n)$ and $\Phi_2(Y_n)$ will contain both p disjoint intervals [5]. Still with inclusion monotonicity of interval arithmetic, since $F_1([1,9]) \cap F_2([1,9]) = \emptyset$, the $2\times p$ intervals obtained will be all disjoint and then $|Y_{n+1}| = |X_{n+1}| = |\Phi(Y_n)| = 2 \times p$.
Moreover, $\Phi(X_0) = \Phi([1,9]) \subset [1,9]$ and thanks to the property 1, we can check that intervals of X_{n+1} and Y_{n+1} are included in $[1,9]$. ▲

Property 3. Bounds of intervals are always maintained in domains. That is to say, if $[a,b]$ is an interval of X_n [6], then $\forall p \geq n$, a and b are interval bounds of X_p.

Proof. First of all, since f_1 and f_2 are monotonous, for any interval I, bounds of $F_1(I)$ and $F_2(I)$ take support on bounds of I. Now, the property is shown by induction: First, bounds 1 and 9 for x and y are always maintained because:
- (x=9, y=9) is a solution of the problem
- x=1 cannot be removed by computing $X \leftarrow \Phi(Y)$ since y=9 is a support.
- y=1 cannot be removed by computing $Y \leftarrow X$ since x=1 is a support.
If we assume now that the bounds of all the intervals in the representation of X_n are maintained for all $p \geq n$, then bounds of X_{n+1} will also be maintained for all $p \geq n+1$ since they take support on bounds of Y_n, i.e. X_n, and they are included in X_n (property 1). ▲

Now, property 1 allows us to say that adding an intersection with domains at each step has no effect. Therefore, this algorithm computes successively projections over x and over y:

$(D_x \leftarrow \Pi_x^{c_1}(B)) \longrightarrow (D_y \leftarrow \Pi_y^{c_2}(B)) \longrightarrow (D_x \leftarrow \Pi_x^{c_1}(B)) \longrightarrow ...$

Property 2 leads immediately to the following fact : The number of intervals in X_n tends to infinity, and even if the size of intervals may tend to zero, each interval contains necessarily one non-removable point (property 3), so we are dealing with an infinity of non-empty disjoint intervals. The AC part of the problem is contained in the result of this algorithm after an infinity of iterations. In a nutshell :

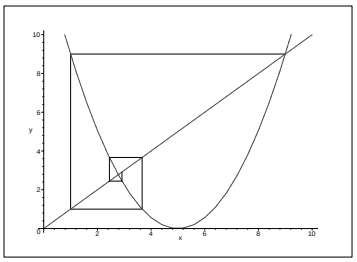
[5] F_1 and F_2 are optimal

[6] We cannot carry on regardless of a bit of rigor here. By saying that $[a,b]$ is an interval of X_n, we mean that $[a,b]$ is an element of an union seen as a set of disjoint intervals. So we consider $[a,b] \in X_n$, and not only $[a,b] \subset X_n$

In the AC part of P, domains are included in an infinity of non-empty disjoint intervals.

Sufficient condition Consider the following numerical series :
$u_0 = 9$
$u_n = f_2(u_{n-1})$

We prove easily that $u_n \rightarrow \frac{25}{9}$ when $n \rightarrow +\infty$. This convergent series is represented on the following picture :



Let A be the set $\{u_n, n \in \mathbb{N}\} \cup \{\frac{25}{9}\}$. Clearly, the *h-box* $X \times Y = A \times A$ is arc-consistent since each point u_n has a support for both constraints. This *h-box* being included in the initial box $[1, 9] \times [1, 9]$ of P, then by definition, the AC part of P contains necessarily this *h-box*.

Now, it suffices to observe that for all n, u_n is exactly a bound of an interval of X_n (a bound "discovered" at the n^{th} step of the algorithm above). Proof is similar to property 2: It comes from the monotonicity of f_1 and f_2, and from the fact that $F_1([1, 9]) \cap F_2([1, 9]) = \emptyset$.

Conclusion We have shown that in the AC part of P, the domain (either for x or for y) includes a set of points u_n (sufficient condition), these points being separated by "gaps" because they are bounds of disjoint intervals (necessary condition). These gaps are inconsistent values that do not belong to the AC part of P. So we have proven the lemma 1.

Remark : In the AC part of P, intervals can be punctual.

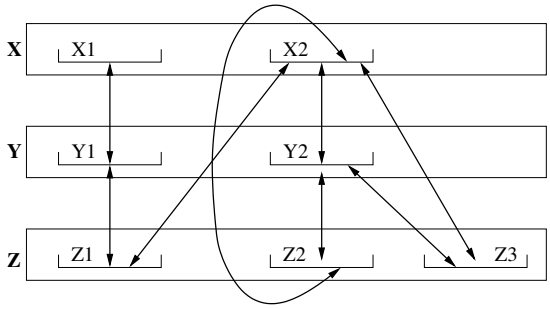
3.2 Box-Set Consistency

We have seen that arc-consistency cannot be achieved over continuous domains. We formally present in this section a stronger consistency that can be achieved.

Let us go back to the example of the previous section, at any step. Let I be an interval of D_x which contains none of the 2 solutions. If we build a box with I and any interval of the current domain D_y, it is not arc-consistent and does not contain any arc-consistent sub-box (see definition 3 and remark 1).

Actually, there are exactly 2 arc-consistent sub-boxes in this example, which are zero-sized boxes around the solutions.

Let us generalize. The following figure depicts a system of 3 variables pairwisely linked by binary constraints. Domains have several intervals, and an arrow between two intervals I and J means that every value of I has a compatible value in J and conversely.



We see that the h-box $(X_1 \cup X_2) \times (Y_1 \cup Y_2) \times (Z_1 \cup Z_2 \cup Z_3)$ is arc-consistent. But there are only two arc-consistent sub-boxes composed with these intervals, which are $X_2 \times Y_2 \times Z_2$ and $X_2 \times Y_2 \times Z_3$.

The box $X_1 \times Y_1 \times Z_1$ is not arc-consistent because X_1 and Z_1 are not linked. Actually, X_1, Y_1 and Z_1 do not belong to any arc-consistent sub-box, and they can be removed from the domains.

In this paper, we present algorithms that find the maximal arc-consistent sub-boxes of a problem.

Definition 6 (Box-set Consistency). *Let $P = (C, V, B)$ be a NCSP. The box-set consistency of P is the set $\{B'\}$ of maximal boxes such that (C, V, B') is an arc-consistent sub-NCSP of P.*

We can either use each of these boxes as a choice point in the original system P (and therefore carry on with splitting), or collect these boxes to get one h-box, which would be $X_2 \times Y_2 \times (Z_2 \cup Z_3)$ in this example.

Box-set consistency is stronger than arc-consistency as the following example illustrates:

Example 5. $D_x = D_y = D_z = [-2, 2] \quad D_w = [1, 4]$
Constraints are $x^2 = w$, $x = y$, $y = z$ and $x = -z$.
Arc-consistency is achieved with the following domains :
$D_x = D_y = D_z = [-2, -1] \cup [1, 2], \quad D_w = [1, 4]$
And box-set consistency discards the whole box (in the domain of x, neither $[-2, -1]$ or $[1, 2]$ belong to an arc-consistent sub-box).

But it is weaker than global solving. It suffices to consider this NCSP:

Example 6. $D_x = [0, 2] \quad D_y = [0, 2] \quad D_z = [0, 2]$
Constraints are $x = y$, $x + z = 2$ and $y = z$.
As the initial box is already arc-consistent, it is box-set consistent. But the real solution is $\{(1, 1, 1)\}$.

3.3 Remark

Box-set consistency can be defined in another fashion. We can characterize arc-consistent boxes as solutions of a problem over the intervals, i.e. a problem where variables take *interval* values instead of *real* values. In this way, Z_1 (in the figure above) would not belong to a solution, and could be discarded as an inconsistent "value". To introduce the definition of this induced problem over the intervals, let us start with a single constraint:

Definition 7 (Arc-Extension of a constraint). $\mathbb{I}$ *represents the set of intervals.*
Let $c(x, y)$ be a constraint relating variables x and y. We call arc-extension of c and denote $arc(c)$ the relation defined on $\mathbb{I} \times \mathbb{I}$ such that:

$$arc(c)(X, Y) \Longleftrightarrow \begin{cases} \forall x \in X,\ \exists y \in Y \mid c(x, y) \\ \forall y \in Y,\ \exists x \in X \mid c(x, y) \end{cases} \tag{1}$$

In a nutshell, $arc(c)(X, Y)$ states that c is arc-consistent. The definition of arc-extension for n-ary constraints is straightforward.

Now the arc-extension of a NCSP is simply defined as follows:

Definition 8 (Arc-Extension of a NCSP). *Let P be a* NCSP *with m constraints $c_1, ..., c_m$ relating n variables $x_1, ..., x_n$ in an initial box $D_{x_1} \times ... \times D_{x_n}$.*
The arc-extension of P is the set of

- *n variables $X_1, ..., X_n$, with $X_i \in \mathbb{I}$*
- *m constraints $arc(c_1), ..., arc(c_m)$*
- *Initial domains : $\forall i\ (1 \leq i \leq n)\ \ X_i \subset D_{x_i}$*

The solutions of this problem are the maximal arc-consistent boxes.

3.4 Number of boxes of a box-set consistent problem

[5] has shown that the number of intervals for a given variable v in a box-set consistent problem is bounded by $((p-1) \times a) + 1) = O(p \times a)$, where p is the maximum number of intervals obtained by *one* projection, and a is the arity of the variable, that is the number of constraints in which v appears. This leads to a total number of boxes of a box-set consistent problem that is bounded by $(p \times a)^n$. In Hyvönen's terminology, this number bounds the size of the *global application space*.

This result holds on problems with only primitive constraints, and without multiple occurrences of a variable in a same constraint.

The result can easily be extended to problems with any type of constraints by considering, instead of n, the number n' of variables in the decomposed system.

On the contrary, the result seems difficult to hold in the general case. Indeed, the box-set consistency of a problem where the variables with multiple occurrences are renamed does not imply the box-set consistency of the (initial) problem, and we have not found straightforward bounds.

4 The natural splitting algorithm

In this section, we show an algorithm that enforces box-set consistency. This algorithm is based on a strategy of bisection called *natural splitting*. Two versions are presented. Both resort to a projection operator that computes unions, but the second version uses this operator only once per box whereas in the first version it is embedded in a propagation loop.

4.1 The key idea

Let us go back to the algorithm of hull-consistency filtering `HC_Filtering` (see 2.3), and assume that this procedure has been applied on a box B. If for every variable the latest projection has produced only one interval, we will show that B is arc-consistent. If one of the last projections produced at least 2 intervals, the idea is to split the domain of this variable into these 2 intervals. This bisection is called *natural splitting*, to contrast with the semantic-less midpoint splitting.

We hope in this way that such a disjunction will not occur anymore on both sub-boxes. We apply the same process on each of the sub-boxes : hull filtering and natural splitting, until we obtain a fix point.

To distinguish constraints whose projections produce 1 interval from those that produce several intervals, we use the term *box-continuity*. We begin by introducing this notion and give the algorithm afterwards.

4.2 Box-Continuity

Box-continuity is the key property of our approach. We will say that a constraint c is *box-continuous* on a given box when projections of c do not create gaps, i.e. when the result set of a projection of c over whatever variable contains a single interval. Formally:

Definition 9 (Box-Continuity). *Let c be a constraint relating variables $x_1, ..., x_k$, B a box.*
c is box-continuous on $B \iff \forall i\ (1 \leq i \leq k)\ |\Pi^c_{x_i}(B)| = 1$.

Box-continuity is not related to the "classical" mathematical definition of continuity for the functions involved in the constraint : for instance, the function $f_1 : (x, y) \longrightarrow x^2 - y$ is continuous on $B = D_x \times D_y = [-2, 2] \times [1, 4]$ whereas $c_1 : f_1(x, y) = 0$ is not box-continuous since $\Pi_x(c_1)(B) = \{[-2, -1], [\ 1,\ 2]\}$.

Conversely, $f_2 : (x, y) \longrightarrow I(x - y)$, where $I(z)$ is the integer part of z, is not continuous on $D = D_x \times D_y = [-2, 2] \times [-2, 2]$ whereas $c_2 : f_2(x, y) = 0$ is box-continuous since $\Pi_x(c_2)(B) = \Pi_y(c_2)(B) = \{[-2, 2]\}$

4.3 First version

To perform natural splitting, we need to know where are the gaps produced by the last projections of `HC_Filtering`. One way to retrieve this information immediately is to modify `HC_Filtering` to allow union labeling. When the domain

of a variable is revised, instead of computing the hull of the projection immediately, we can keep a union and computes the hull only when this domain is used as a parameter of another projection. Without changing anything to the algorithm, this trick provides a way to detect box-continuity very easily. Indeed, once hull-consistency is achieved, if the domain of every variable is a single interval, it means that the latest projection performed over any variable resulted in a unique interval, i.e. that all the constraints are box-continuous (on the box).

The following procedure is the first version of our algorithm. It applies the "union" variant of `HC_Filtering`, and split the box as long as a domain in the box contains a gap:

Procedure 2 `Naive_BoxSet`(NCSP (C, V, B), in-out `solutions`)

```
 2: for all pairs < c, x > in C × V do
      if x is related by c then
 4:     add < c, x > in Q
    while Q is not empty do
 6:   pop a pair < c, x > from Q
      D'_x ⟵ Π^c_x(□D_{x_1} × ... × □D_{x_n})
 8:   if (□D'_x ⊂ □D_x) then
        Propagate((C, V, B), Q, c, x)
10:     D_x ← D'_x

12: if (exists a variable x_i with |D_{x_i}| > 1) then
      for j = 1 to |D_{x_i}| do
14:     B ← □D_{x_1} × ... × □D_{x_{i-1}} × D^j_{x_i} × □D_{x_{i+1}} × ...□D_{x_n}
        Naive_BoxSet((C, V, B), solutions)
16: else
      if (B is not empty) then
18:     add B to solutions
```

Lines 2-10 are the union variant of `HC_Filtering`. Lines 12-15 perform natural splitting.

4.4 Properties

Consider, in the execution of `Naive_BoxSet`, the point where the box B is added to solutions, i.e. at line 18. If we have reached this point, it means that no gap could be found in B, or in other words, that every constraint is box-continuous on B. But B is also hull-consistent so the following proposition applies to B:

Proposition 1. *If every constraint is box-continuous on a box B then:*
B is hull-consistent $\iff$ B is arc-consistent

Proof. Once the fix point of a hull consistency filtering is reached, we have for every pair $< c, x >$ of constraint/variable: $D_x = \Box\Pi^c_x(B)$. As c is box-

continuous, $\square\Pi_x^c(B) = \Pi_x^c(B)$ and then $D_x = \Pi_x^c(B)$, which means that the domain of x is arc-consistent regarding c. The converse relation is obvious. ▲

Hence, B is an arc-consistent box. As a rule of thumb: *Hull consistency and Natural Splitting gives the box-set consistency of the problem.*

4.5 Lazy version

In practice, managing unions in `HC_Filtering` is highly inefficient. Moreover, results of projections are computed all along the propagation loop although we are interested only by the last ones. Imagine now that we got a way to check quickly whether a constraint is box-continuous or not. We could apply `HC_Filtering` as it is (without unions), and once the fix point is reached check the constraints one after the other until we find a constraint c that is not box-continuous. If one is found, there is at least one variable x involved in c for which we can exhibit a gap inside the domain. So we compute an exact projection this time to disclose the gap, and finally use it as a candidate for splitting.

So, our solving strategy now is simply a combination of three steps: hull-consistency filtering, gap search, and natural splitting:

Procedure 3 `Lazy_BoxSet`(NCSP (C, V, B), in-out `solutions`)

```
 2: HC_Filtering((C, V, B))

 4: if B is empty then
      return
 6: C' ← C
    found ← false
 8: while (not found) and (C' ≠ ∅) do
      pop c from C'
10:   if c is not box-continuous then
        V' ← the set of variables in V related by c
12:     while (not found) and (V' ≠ ∅) do
          pop x from V'
14:       U ⟵ Π_x^c(B)
          if |U| > 1 then
16:         found ← true

18: if found then
      for all intervals I in U do
20:     D_x ⟵ I
        Lazy_BoxSet((C, V, B), solutions)
22: else
      add B to solutions
```

Line 2 in `Lazy_BoxSet` enforces a hull consistency filtering. In lines 6 to 16, we try to find a gap in the box. In case of success, lines 19 to 21 execute a natural split, otherwise, the box is stored in solutions (lines 23).

4.6 Detection of Box-Continuity

With an implementation of `HC_Filtering` like `HC4` [2], it is easy to detect box-continuity of a constraint during the filtering step of `Lazy_BoxSet`.

The projection operator of `HC4` must be slightly modified to update this property while exploring the syntax tree of a constraint. The rule is simple: before projecting a constraint c, the projection operator sets the box-continuity boolean of c to `true`. If a disjunction appears somewhere in the tree, this boolean is set to `false`.

Thus, in practice, detecting box-continuity is computationally insignificant and permits to dramatically reduce the number of calls to the general projection operator embedded in `Lazy_BoxSet`. This remark is relevant for all the problems, because at some point in the search, a majority of boxes are small enough for functions to be all monotonous. It is straightforward that with monotonous functions, detection of box-continuity always succeeds so that the projection operation is not costly.

4.7 Difference with Hyvönen's method

In [5], natural splitting is also used in a similar solving strategy. But an important difference makes our version much more powerful. In [5], no hull filtering is used before splitting and only box-continuous constraints are projected before instantiating variables. In a majority of non-linear problems, this extremely decreases the performances by generating an exponential number of "overlapping situations" [5], as this example illustrates:

Example 7. Let $P = (C, \{x_1, ..., x_{50}, y\}, D_{x_1} \times ... \times D_{x_{50}} \times D_y)$ be a NCSP.
$D_{x_1} = ... = D_{x_{50}} = [-2, 2]$ and $D_y = [1, 4]$.
C includes the following constraints:
$x_1^2 = y$
...
$x_{50}^2 = y$
Finally, C contains a trivially unsatisfiable constraint: $x_1^2 = -x_1^2$

We assume that constraints are treated in their declaration order. For all i $(1 \leq i \leq 50)$, projection of $x_i^2 = y$ over x_i gives $\{[-2, -1], [1, 2]\}$ so that `HC_Filtering` will not perform any reduction (bounds of $[-2, 2]$ are preserved). After these 50 unfruitful projections, `HC_Filtering` will fall on the last constraint, that makes the whole box inconsistent. So `Lazy_BoxSet` terminates almost immediately.

In contrast, as no constraint is box-continuous, the method in [5] will introduce a choice point for every variable x_i and deploy a search tree of 2^{50} leaves

before detecting inconsistency. A combinatorial explosion occurs. We observed this difference with simple problems of distance equations. In the general case, the proposed algorithm is more costly than our naive version.

Another drawback is that the domain of a variable must be divided statically into sub-intervals (the *actual application space*) where constraints are all box-continuous. This computation is only possible with primitive constraints.

5 Conclusion

We have tried to put an end to the question of arc-consistency with continuous domains, by showing precisely on a simple example that it is not applicable.

However, we have given a way to obtain the *box-set consistency*, i.e. all the arc-consistent sub-boxes of a problem, using a new splitting strategy called *natural splitting*.

The `Lazy_BoxSet` algorithm enforcing box-set consistency has been implemented, and so far, validated on toy problems. This implementation includes a projection operator for handling any type of constraints (not only primitive constraints). We will discuss about this crucial operator in a future paper, along with the conditions under which it can be applied.

Beyond this implementation, we believe that box-set consistency is a strong hence interesting property. We are currently investigating possible combinations of box-set filtering and interval analysis.

References

1. F. Benhamou. Interval constraint logic programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends, LNCS no 910*, pages 1–21. Springer Verlag, 1995.
2. F. Benhamou, F. Goualard, L. Granvilliers, and J-F. Puget. Revising hull and box consistency. In *International Conference on Logic Programming*, pages 230–244, 1999.
3. F. Benhamou, D. McAllester, and P. Van Hentenryck. Clp(intervals) revisited. In *International Symposium on Logic programming*, pages 124–138. MIT Press, 1994.
4. B. Faltings. Arc-consistency for continuous variables. *Artificial Intelligence*, 65, 1994.
5. E. Hyvönen. Constraint reasoning based on interval arithmetic—The tolerance propagation approach. *Artificial Intelligence*, 58:71–112, 1992.
6. O. Lhomme. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles*. Phd thesis, University of Nice-Sophia Antipolis, 1994.
7. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
8. R. Moore. *Interval analysis*. Prentice-Hall, 1977.
9. D.L. Waltz. Understanding line drawings of scenes with shadows. *The Psychology of Computer Vision*, pages 19–91, 1975.

Implementing explained global constraints

Étienne Gaudin[1], Narendra Jussien[2], and Guillaume Rochart[2,3]

[1] Bouygues e-lab
1 av. Eugène Freyssinet – F-78061 St Quentin en Yvelines Cedex
[2] École des Mines de Nantes, Département Informatique
4, rue Alfred Kastler - B.P. 20722 – F-44307 Nantes Cedex 3
[3] LINA, FRE CNRS 2729
2, rue de la Houssinière – B.P. 92208 – F-44322 Nantes Cedex 3
egaudin@bouygues.com,{jussien, grochart}@emn.fr

1 Introduction

Constraint satisfaction problems (CSP) [22] have proven to be an efficient model for solving many combinatorial and complex problems. Moreover, recurring patterns and sub-problems in those problems are now handled through global constraints [2, 4, 18]. Global constraints have been a good advocate for using constraint programming techniques to solve real-life problems.

Explanations (specializing (A)TMS [7]) and generalizing nogoods [21]) have been initially introduced to improve backtracking-based algorithms [8]. However, they have been recently used for many other purposes [11] including new solving techniques [13, 14], dynamic constraint solving [23, 6] and user interaction [15]. Explanations represent an explicit and limited *trace* of the behavior of the solver. They are mainly used both for dynamic solving and user-interaction purposes. Maintaining and computing explanations may be particularly costly. Fortunately, explanation-based algorithms efficiently solve problems thanks to mechanism reducing *thrashing* during the resolution.

Introducing both explanations and global constraints in a constraint solver is a real challenge. Indeed, three points may be kept in mind: (1) providing precise and meaningful explanations to be offered to the user extracting the locality of part of the reasoning of the global constraint, (2) altering the efficiency of the original constraint as few as possible *i.e.* developing algorithms within the constraints efficient for both filtering and explanation capabilities and (3) providing algorithmic tools for both incrementality (taking into account new events from filtering) *and* decrementality (in order to be able to replace backtracking with repair-based techniques as explanation-based solvers usually offer).

In this paper, we illustrate those three points with a global constraint encapsulating flow theory that maintains a feasible flow in a given network, similar to the flow constraint in [5]. Such a constraint is useful for several real-life problems involving resource allocation and is well suited for illustrating the challenges of embedding explanations. In the following, we first set the context of our study in Section 2 and introduce the flow constraint (Section 3). Then, the three points are successively addressed in Sections 4, 5 and 6 keeping the flow constraint as an illustration. Finally, some experiments are introduced showing the interest and

the efficiency of our approach in Section 7 with respect to the three challenges defined below.

2 Context

2.1 Constraint Satisfaction Problems

Following [22], a *Constraint Satisfaction Problem* is made of two parts: a syntactic part and a semantic part. The syntactic part is a finite set V of variables, a finite set C of constraints and a function $\text{var} : C \to \mathcal{P}(V)$, which associates a set of related variables to each constraint. Indeed, a constraint may involve only a subset of V. Constraints are defined by their set of allowed tuples providing the semantic part of the constraint network. A tuple is a set of couples (var, val) where $var \in \text{var}(c)$ and $val \in d_{var}$ its domain (set of allowed values for var). Notice that domains are considered here as unary constraints.

2.2 Global constraints

In practice, constraints are seldom provided as a set of allowed tuples. Usually, arithmetic or symbolic expressions are used to describe the constraints of a given CSP. Moreover, *global constraints* [2, 18] are now often use to encompass several recurring patterns arising in optimization problems. Following [4] we will focus here on operationally global constraints: *Operational globality considers both a constraint and a consistency notion. The constraint is said to be* global *if there exists no decomposition scheme for which the consistency notion removes as many local inconsistencies as on the original constraint. This concept is important because it compares the pruning of the constraint and its decompositions* wrt *a consistency notion. If the constraint prunes more than its decompositions, then, from a CSP standpoint, it is operationally global, since the closure of the decomposition cannot recover the consistency achieved on the original constraint.*

Such a constraint, from an implementation point of view, usually involves specific filtering algorithms that maintain a given support structure[4] in order to provide useful information for powerful domain reductions or early identification of failures. For example, for the `allDifferent` constraint [17], a reference matching is usually maintained which is used to compute strongly connected components for the sake of the filtering algorithms.

2.3 Explanations for constraint programming

An explanation [11] for constraint programming contains enough information to justify a decision (throwing a contradiction, reducing a domain, etc.): it is composed of the constraints and the choices made during the search which are sufficient to justify such an inference.

[4] Notice that the support structure may not be explicitly maintained within the constraint but algorithmic tools are usually provided in order to be able to access it in a reasonable time.

Definition 1. *An **explanation** of an inference ($\mathcal{X}$) consists of a subset of original constraints ($\mathcal{C}' \subset \mathcal{C}$) and a set of instantiation constraints (choices made during the search: $d_1, d_2, \ldots, d_k$) such that: $\mathcal{C}' \wedge d_1 \wedge \ldots \wedge d_n \Rightarrow \mathcal{X}$.*

An explanation e_1 is said to be more precise *than explanation e_2 if and only if $e_1 \subset e_2$.*

The more precise an explanation, the more useful it is. However, this is not a sufficient condition. There can be numerous minimal explanations. This is why we have chosen to find a compromise between precision and ease of computation.

Computing explanations The most interesting explanations are those which are minimal regarding inclusion. Those explanations allow highly focused information about dependencies between constraints and variables. Unfortunately, computing such an explanation can be exponentially costly. A good compromise between precision and ease of computation consists in using the solver embedded knowledge to provide explanations [12]. Indeed, constraint solvers always know, although it is scarcely explicit, *why* they remove values from the domain of the variables. By making that knowledge explicit, quite precise and interesting explanations can be computed as constraint solvers are supposed to efficiently perform their task! Therefore, explanations strongly depend both on the constraint solver at hand and on the way the problem is modelled.

2.4 User accessible explanations

The primary usage of explanations is to provide the user some feed-back about resolution. Precise and meaningful explanations are therefore required. Moreover, some kind of user interface for explanations is also required. Indeed, explanations as we introduced them, reflecting the behavior of the solver, may be far from the user representation of the problem. Tools are needed to translate that internal information to a problem related representation [15].

2.5 Explanations for dynamic constraint solving

Solving dynamic constraints problem has led to different approaches [23]. Two main classes of methods can be distinguished: proactive and reactive methods. On the one hand, proactive methods propose to build robust solutions that remain solutions even if changes occur. On the other hand, reactive methods try to reuse as much as possible previous reasonings and solutions found in the past. They avoid restarting from scratch and can be seen as a form of learning. Using explanations falls is such a reactive techniques [11, 6].

Explanations for dynamic constraint retraction Dynamic constraint retraction is basically a two-step process [6]: enlarging the current environment (in order to *undo* past effects of the retracted constraint which are the events whose explanation contains the retracted constraints) and restoring a given consistency for the resulting constraint network.

Search as a dynamic process Using explanations to help solving (as in *Dynamic Backtracking* [8] or in `mac-dbt` [13]) modifies the way search is done. In those algorithms (as in `decision-repair` [14]) backtracking is replaced by a repair mechanism as handling contradiction is no longer handled by getting back to an already explored safe situation. Contradiction can now be precisely explained allowing modifications in the set of decisions (usually variable assignments) performed in a chirurgical way.

Those recent algorithms introduce new events in a constraint solver (domain enlargement) and a new kind of incrementality: decremental requirements.

2.6 Global constraints and explanations: operational requirements

Implementing global constraints in an explanation-aware setting leads to three challenges:

- *computing (quasi-)minimal and user meaningful explanations.* Explanations computing for and by a global constraint should provide as much information as possible *ie*, as paradoxal as it may appear, explanations should be as local as possible. Moreover, they should be easily interpreted in terms of the underlying theory of the constraint in order to be translated into comprehensible terms for a user. Consider for example the `allDifferent` constraint where a value is removed from the domain of a variable when the variable and its value are in two different strongly connected components in the graph representing the current reference matching. In order to provide an explanation for that situation some more information need to be computed and a theoretical analysis should be performed [20]. Usually, the support structure need to be enhanced in order to provide efficient tools for efficient explanations during propagation[5].
- *efficiency of the constraint must be preserved.* Adding explanations capabilities into a global constraint must not degrade its performance to an unbearable level. The key point for such constraints is to *incrementally* maintain the enhanced support structure. Recomputing from scratch the information upon each value removal should be avoided in order to maintain a low computational cost of the filtering algorithms. Indeed, the idea is to be able to design a propagation algorithm efficient both for filtering and explanation computation.
- *incrementality and decrementality should be provided.* As with explanations, backtracking can easily be replaced with some kind of repair mechanisms, constraints should be able to provide both incremental AND decremental algorithms need to be designed in order to provide efficient and powerful constraints for new search algorithms.

In the following, we will illustrate those three points with a global constraint that maintains a feasible flow in a given network.

[5] Notice that this is not always the case: see the `stretch` constraint for example[20].

3 Application to the flow Constraint

We chose here to illustrate the way global constraints should be instrumented for explanation-based solvers with a `flow` constraint. Such a constraint is meant to maintain a feasible flow in a given network. We recall some information about flow theory and introduce the global constraint.

3.1 Flow Theory

The subsection content is based on the book "Network Flows" by Ahuja, Magnati and Orlin [1]. We only give a quick survey of the main definitions and properties used in this paper. The interested reader will find much more details and all proofs in it.

Let's note $G = (N, A)$ an acyclic directed graph with N the set of nodes and A the set of arcs.

Definition 2 (Feasible Flow). *The following equations model a feasible flow problem on G between two specifics nodes of N, the source s and the sink t:*

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ij} = \begin{cases} v \text{ for } i = s \\ 0 \ \forall i \in N \backslash \{s,t\} \\ -v \text{ for } i = s \end{cases} \tag{1}$$

$$\forall (i,j) \in A, l_{ij} \leq x_{ij} \leq u_{ij}, l_{ij} \geq 0 \tag{2}$$

Equations (1) ensure flow conservation at each node, (2) impose that the amount of flow going through each arc is between its lower bound l_{ij} and upper bound u_{ij}. We refer to $x = \{x_{ij}\}$ satisfying (1) and (2) as a feasible flow and the value of v as the value of the flow. Figure 1 gives an example of a feasible flow problem.

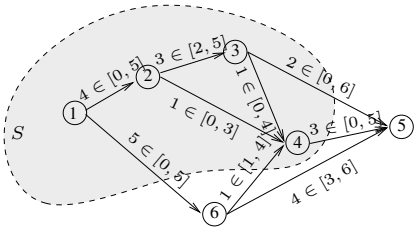


Fig. 1. Flow example in an acyclic oriented graph

Let's now introduce cuts and their capacity. A cut is a partition of the set of nodes N into two subsets. Flow theory is interested by a specific type of cut, the $s - t$ cut.

Definition 3 ($s - t$ Cut). *An $s - t$ cut $(S, \bar{S})$ is a partition of the set of node N into two subsets S and $\bar{S} = N - S$ such as $s \in S$ and $t \in \bar{S}$.*

A cut is denoted by $(S, \bar{S})$, we use also this notation to refer to the set of arcs (i, j) such as $i \in S$ and $j \in \bar{S}$. The capacity of a cut gives bounds on the amount of flow we can send from one set of the nodes to the other one. It plays a central role for explanation in a flow constraint (see Section 4).

Definition 4 (Capacity of an $s-t$ Cut).

$$C(S, \bar{S}) = \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(i,j)\in(\bar{S},S)} l_{ij} \tag{3}$$

For example, in Figure 1, the capacity of the cut $(\{1, 2, 3, 4\}, \{5, 6\})$ is 15. It is equal to the difference between the sum of the maximum capacities $u_{16} = 5$, $u_{45} = 5$ and $u_{35} = 6$ and the minimum capacity $l_{14} = 1$.

Let us introduce two properties linking cut capacity and flow value v.

Property 1 *The value v of any flow x is less than or equal to the capacity of any $s-t$ cut in the network*

This property is quite intuitive as any flow should pass through every $s-t$ cuts and therefore can not exceed their capacity.

Property 2 (Max-Flow Min-Cut) *the maximum value v of the flow x from s to t equals the minimum capacity among all $s-t$ cuts.*

This last property shows the dual aspects of flow problems, any maximum flow problem is equivalent to a minimum cut problem.

3.2 Semantics

The flow constraint studied in this paper is a direct translation in a global constraint of the feasible flow problem (see definition 2). Compared to Chip flow constraint [5], it is restricted to flow feasibility on acyclic graph. The following call `flow(pb,G,s,t,V)` in a Choco program [16] defines a flow on G between s and t of value v where:

- `G` is a list formulation of the graph G; an arc (i, j) is defined by couple `(j,`$\mathtt{X_{ij}}$`)` in the list `G[i]` where $\mathtt{X_{ij}}$ is the variable denoting the flow going through the arc $(i, j) \in A$ with its domain $d_{\mathtt{X_{ij}}}=[l_{ij}..u_{ij}]$,
- `s` and `t` are the index of the source and sink nodes s and t in the list `G`,
- `V` is the variable denoting the flow value v between source and sink,

The feasible flow given in Figure 1 corresponds to the following Choco call.

```
Problem pb = new Problem();
IntVar X12 = pb.makeEnumIntVar("X12", 0, 5);
...
CapaEdge[][] G = new CapaEdge[][]{
             {new CapaEdge(X12,2), new CapaEdge(X16,6)}, //(1,i) arcs
             {new CapaEdge(X23,3), new CapaEdge(X24,4)}, //(2,i) arcs
...
pb.post(flow(G, 1, 5, V));
```

The following section presents propagation principles and flow algorithms used in the `flow` constraint.

3.3 Flow Algorithms

To implement a Flow constraint, propagation algorithms should answer the following questions:

1. Is the support set empty or not?
2. What are the bounds of `V` according to `G`?
3. What are the bounds of $\texttt{X}_{\texttt{ij}}$ of `G` according to `V` or other $\texttt{X}_{\texttt{ij}}$ variables?

The first point is equivalent to solving the Feasible Flow Problem of (2). The two last points are related to Minimum and Maximum Flow Problems with Minimum Capacity.

Feasible Flow Problem Finding a feasible flow is obvious when there is no minimum capacity on arcs (*i.e.* $\forall (i,j) \in A, l_{ij} = 0$). For such a graph an empty flow is a feasible one. When some l_{ij} are greater than zero, Berge gives in [3] an algorithm based on solving a maximum flow problem on a transformed graph. This graph has two interesting property:

1. a feasible flow x exists in G if and only if the value of a maximal flow in the transformed graph saturates arcs of its source and sink;
2. all arc's minimum capacity in the transformed graph is zero so standard maximum flow algorithm could be applied.

This method is used straightforwardly to find a feasible flow or to prove that the support of a `flow` constraint is empty. In the rest of the paper, we use Φ to denote a support, *i.e.* a feasible flow of G.

Minimum and Maximum Flow Problem with Minimum Capacity The existence of minimum capacity on arcs makes computing minimum or maximum flow more complex. A two stage algorithm should be applied:

1. find a feasible flow Φ using the previous algorithm;
2. applying a maximum flow algorithm on the residual network[6] containing Φ.

For finding a maximum flow, one just has to solve a maximum flow problem on the residual network containing Φ. Finding minimum flow is also quite simple. One has to solve a maximum flow problem on the same residual network but from the sink t to the source s. Intuitively, this last problem removes as much flow as possible between t and s while maintaining a feasible flow. Finally, the resulting flow is feasible and minimum.

[6] The residual network is a common graph representation that contains both the flow and the remaining flow of each arcs. It is used in most graph algorithms to maintain incrementally a flow.

Propagation principles Two types of propagation have been defined in the `flow` constraint:

1. **Flow conservation propagations** *inside* $\{\mathtt{X_{ij}}, \forall (i,j) \in A\}$ *and between* $\{\mathtt{X_{ij}}, \forall (i,j) \in A\}$ *and* `V`
   These propagations are local to each node. They insure the flow conservation constraint (1) by applying the standard (in CP solver) bound propagation rules of linear formula [9].
2. **Global flow propagations** *from* $\{\mathtt{X_{ij}}, \forall (i,j) \in A\}$ *to* `V`
   They are global to the graph and based on algorithms described below. They check the existence of a feasible flow Φ (*i.e.* a support) and maintain the value of the flow in the graph, *i.e.* the domain of the `V` variable, by solving a minimum and a maximal flow problems[7].

A small modification of the original graph G is used to improve propagations without any change in the algorithms. We add a new node, denoted st for super sink, with one arc from sink t to this super sink st with variable `V` as the arc domain capacities. This transformation imposes that any feasible flow Φ should be in the domain of `V` and so insures bound consistency of the `V` variable.

The propagation algorithms are combined in the flow constraint according to the complexity of the underlying algorithms. Here is a sketch of the algorithms. Propagations based on standard linear equation propagations are called first. After reaching a fix-point without contradiction, the global flow propagations is called. These last propagations compute new bounds of the flow variable `V`. If they are different from current ones, two situations could occur: if the computed bounds are inconsistent with current bounds of the flow variable `V`, a contradiction should be triggered; if the computed bounds reduce the domain of variable `V`, the new bounds should be propagated.

We detail in Section 5 how this process could be speed up using a incremental version of the global flow propagations.

4 Accurate explanations for the `flow` constraint

Explaining efficiently global constraints is a challenging task: generated explanations must be as precise as possible without slowing down the resolution. This section presents how to generate precise explanations for the `flow` constraint thanks to nearly cost less algorithms based on the cut notion.

Since the `flow` constraint propagates on variables bounds, we will note $expl(x \geq v)$ (resp. $expl(x \leq v)$) the explanation of the lower bound of variable x and more precisely the reason why the variable x must be at least equal to v (resp. the reason why x must be less than or equal to v). With respect to the `flow` constraint, two kinds of filtering removals are deduced: the flow conservation rules and the minimum, maximum or feasible flow properties.

[7] We choose not to solve a minimum and a maximum flow problems on each arc to maintain $\mathtt{X_{ij}}$ bounds as it seems to be too costly. These bounds are only update by the flow conservation propagations.

4.1 Flow conservation explanations

As illustrated on equation (1), flow conservation rules are based on the Kirchoff law. Since flow in the network is considered as positive values on the arcs, the following rule can be used : $\forall i, k \in N \backslash \{s,t\}, (i,k) \in A, u_{ik} = \sum_{j:(j,i)\in A} u_{ji} - \sum_{j\neq k:(i,j)\in A)} l_{ij}$.

This rule makes the explanation explicit: the lower and upper bounds involved in this equation imply the value of the new upper bound of x_{ik}. The explanation of this bound is then $(\forall i, k \in N \backslash \{s,t\}, (i,k) \in A)$:

$$expl(\mathtt{X_{ik}} \leq u_{ik}) = \bigcup_{j:(j,i)\in A} expl(\mathtt{X_{ji}} \leq u_{ji}) \cup \bigcup_{j\neq k:(i,j)\in A} expl(\mathtt{X_{ij}} \geq l_{ij}) \quad (4)$$

This result can easily be generalised to all filtering rules based on this law.

4.2 Maximal flow explanation

As we are dealing with global properties, computing precise explanations is not easy. We could actually use the trivial explanation (union of all variables explanations in the network) but this is far from being precise.

In order to keep things clear, all flows are assumed to be feasible in this section. Properties 1 and 2 guarantee that the maximum value of the flow from a source s to a sink t equals the minimum cut capacity. Computing such a cut is nearly costless as soon as a maximal flow is found. Thus a minimum capacity cut can be used to explain the maximal flow in a network. The capacity of a cut is defined on Equation 3: $C(S,\bar{S}) = \sum_{(i,j)\in(S,\bar{S})} u_{ij} - \sum_{(i,j)\in(\bar{S},S)} l_{ij}$.

The equation shows that the only way to increase the upper bound of the flow across the network is to modify one of these bounds. The following property is now immediate :

Proposition 1 *Given a network and a minimal $s-t$ cut $(S,\bar{S})$ in this network, an explanation for the maximum flow across the network can be defined as follows (where $expl(\mathtt{X_{ij}} \leq u_{ij})$ and $expl(\mathtt{x_{ij}} \geq l_{ij})$ are maintained bound explanations):*

$$expl(\mathtt{V} \leq C(S,\bar{S})) = \bigcup_{(i,j)\in(S,\bar{S})} expl(\mathtt{X_{ij}} \leq u_{ij}) \cup \bigcup_{(i,j)\in(\bar{S},S)} expl(\mathtt{x_{ij}} \geq l_{ij}) \quad (5)$$

For instance, in the network illustrated on Figure 1, when a maximal flow is reached, the cut $S = \{1\}$ is computed. This means that the explanation should contain the upper bound explanations for$x_{1,6}$ and $x_{1,2}$. Such an explanation is sufficient for justifying the new upper bound of the flow across the network from 1 to 5 – here this upper bound equals $u_{1,5} + u_{1,6} = 10$. The computation of such an explanation is quite easy when a maximal flow is known: the algorithm compute reachable nodes from source in the residual graph, and the cut contains all arcs between these nodes and nodes outside ($O(m \cdot n)$ complexity). This algorithm does not guarantee minimal explanations, but it provides an algorithm with an acceptable computing overhead. Since explanations can be exploited to dynamically remove constraints, they need to be always available for each variable domain. This makes the use of algorithms like QuickXPlain [10] (based on dichotomic search of minimal conflict set over constraints) prohibitive.

The same principles are used to compute explanations for minimum flow.

4.3 Feasible flow

When there is no feasible flow in the graph, a contradiction is raised. This situation needs to get an explanation. As we saw earlier, searching for a feasible flow is the same as searching for a maximum flow in a modified graph. Therefore, the same idea applies. The only issue is to take into account the modifications the graph in the explanations (see [19] for details).

5 Efficient filtering algorithm and explanation algorithm

The main difficulty of the implementation of a global constraint lies in the dynamic nature of CP solvers. Constraints must react to solver events such as modifications of the domain of their variables, in order to check their feasibility and deduce some domain reductions or inconsistency. A keen constraint implementation must trigger as less propagation rules as possible for any set of events. A solution is to filter solver events with the constraint support structure and to use incremental algorithms.

5.1 Filtering events with flow constraint support structure

The `flow` constraint reacts only to bound modification events of its variables: an increase/decrease of the lower/upper bound of an arc capacity (variables $\{\mathtt{X_{ij}}\}$) and an increase/decrease of the minimum/maximum flow (variable `V`). For both events, we use the same two stage mechanism[16]: flow conservation propagations rule is triggered immediately and global flow propagations are delayed.

For such delayed propagations, triggered events are filtered through constraint support structure. For the `flow` constraint, we choose a residual network as the support structure. At any fix-point, it contains a feasible flow Φ. Two reactions can happen according to Φ:

1. The new domain bounds are consistent with current support flow Φ, *i.e.* x_{ij} the flow on arc (i, j) in Φ is still in the reduced domain $d_{\mathtt{X_{ij}}}$. Triggering a propagation rule is useless, as Φ is still a support.
2. It is incompatible with current flow Φ, post a new event to the solver in order to awake the constraint for finding new feasible flow and, if necessary, computing the new minimal/maximum flow.

5.2 Finding incrementally a new feasible flow

The second item is done incrementally. The basic principle is to start from current flow support Φ instead of starting with empty flows. We suppose that making the few changes to obtain a new valid support is less costly than starting from scratch.

As explained in Section 3.3, the graph transformation of Berge [3] was designed to search a feasible flow from a null flow by adding new arcs in order to fill a lack of flow in some nodes and to remove an excess of flows in some other

ones. It can be generalised to add or remove some flows on any arcs from any initial flow and so, used to change any flow into a new one consistent with new bounds on arc capacities.

To build the transformed graph, one should start from any feasible flow and:

- initialise $\forall i \in N, b(i) = 0$ a b vector representing lack or excess of flow at each node;
- in order to add q units of flow on arc (i, j) (because the lower bound is more than the support value), remove q from $b(i)$, add q to $b(j)$ and update the structure;
- in order to remove q units of flow on arc (i, j) (because the upper bound is less than the support value), add q to $b(i)$, remove q from $b(j)$ and update the structure.

When all the updated flows have been taken into account, each node i with $b(i)$ positive (resp. negative) is linked to a new source node (resp. sink node) with a maximum capacity on the arc equal to $b(i)$.

A feasible flow compatible with all the added and removed flows exists if and only if maximum flow saturates source and sink arcs. The proof of this property is a direct extension of the one used in [1] for the feasible flow.

To use this algorithm for an incremental version of propagation rules, one has to add in the support structure a b vector. For all events on the bounds of $\mathtt{X_{ij}}$ and $\mathtt{V}$ variables, the flow constraint should update the support graph and, if this new bound is inconsistent with current flow Φ, the b vector.

Each filtering decision must be precisely explained. As Section 4 shows, computing explanations depends on cut. Fortunately this is nearly costless as soon as a maximum flow (in G or G') is available thanks to the Φ support.

6 Decremental data structures

Incremental algorithms presented in last section depends on up-to-date data structures. CP solvers provide backtracking mechanisms that maintain past states of the structure. In dynamic solving (like explanation based algorithms) decremental process must be included into the constraint to ensure the structure to be always consistent with the current state of variables.

6.1 Classical solving

All information depending on the state of variables used by the filtering algorithm must be up-to-date. This compels the following data to be maintained whenever the state is modified:

- the support Φ (a feasible flow), since all filtering algorithms depend on them;
- all data depending on the state of variable, like b values.

With classical backtrack, the state reached after a contradiction is always a past state. This means that the data structure only needs to be stored before trying new search decisions: classical trailing method is sufficient.

6.2 Dynamic solving

In a dynamic context (where a constraint or a decision can be undone even if this is not the last posted one), trailing is useless as the state reached after a contradiction is not necessarily a past state anymore. That means that the constraint must be able to update itself its data structure upon repairing.

With the `flow` constraint, removing constraints does not make the current data structure false, since a feasible flow in a network is still feasible in a less constrained network. This means that in many cases, the constraint only has to update maximum flow and minimum flow from past supports.

However, in 5.1 we saw that the support is not always up-to-date since compatibility check of the support is delayed for the sake of efficiency. Thus it is not possible to use the current support to build a new one. The following algorithm scheme is used to overcome this issue:

- *when an event is triggered, the event is stored in a stack;*
- *when a contradiction occurs, the stored support is restored and updated according to stored events;*
- *when a feasible flow is found, the support is stored and the stack cleaned.*

Similar methods may be useful for all constraints involving delayed filtering rules. It guarantees that the obtained structure is consistent with the new state by updating support with past events, while still using incremental filtering algorithms.

7 Experiments

7.1 Problem presentation

Main constraints The `flow` constraint is particularly handy to model resource affectation problems. This is why an employee scheduling problem – inspired from a Bouygues staff management problem – is presented here. In this employee scheduling problem, one wants to schedule team of employees in a weekly planning. Two concerns should be affected:

- a *day off*: each team has a day off between Monday and Saturday;
- a *shift*: each team works according to predefined shifts.

The problem must checks the company loads are respected – enough teams should work according to the load of work – *and* the assignments should be fair – a team cannot be affected to the same shift or day off too many times.

For example, the bounds illustrated on Table 1 are used for the bench in the next section. In this instance, a team cannot be affected more than 3 times to the same day off, 2 teams should be affected to the evening shift each week, etc.

Additional constraints To force equitable affectation, some other sequencing constraints are added. For instance, in this bench, a team cannot be affected to the evening shift during two consecutive weeks. In real life problems, other constraints such as each team is affected to the Saturday day off at least every five weeks, must be added.

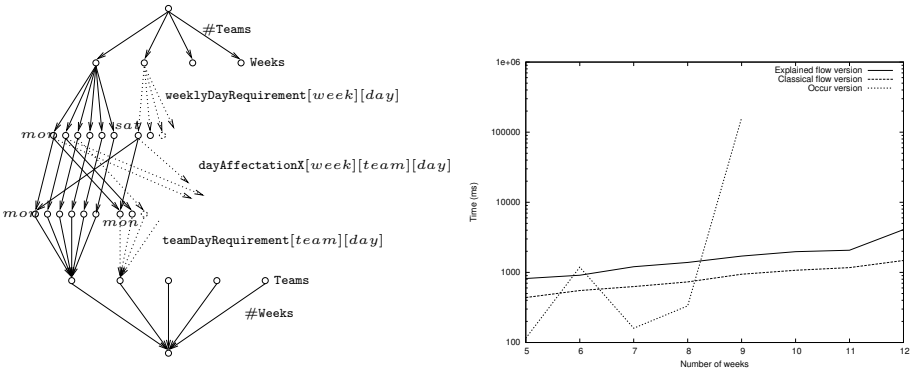
Table 1. A simple instance of an employee scheduling problem

	Day affectation						Shift affectation				
	mon	*tue*	*wed*	*thu*	*fri*	*sat*	*early*	*morning*	*day*	*afternoon*	*evening*
Team min.	0	0	0	0	0	0	0	0	0	0	0
Team max.	3	3	3	3	3	3	4	4	4	4	5
Week loads	1	1	1	1	1	1	1	1	1	1	2

7.2 Problem modelling

This problem can be modelled with numerous basic `occurrence` constraints. However the problem of affecting both days off and shifts can be considered as two independent `flow` constraints. Indeed let `Teams` be the set of teams of employees, `Weeks` the set of all weeks in the problem, `dayAffectationX` variables stating if a day is affected given a week and a team, and `weeklyDayRequirement` and `teamDayRequirement` affectation bounds for each day.

Then the problem of finding a day affectation for each team and week is equivalent of finding a flow in the network illustrated on Figure 2(a). Indeed, the first level of nodes constrains that each week #`Teams` days off must be affected (this means that each team must have a day off per week). The second level constrains the number of days off that can be affected depending on the day in the week (for instance if the company has a lot of work on Monday, no day off should be affected on Monday). The third level is composed of affectation variables modelled as presented before. And the fourth and fifth levels are the symmetric levels for teams requirement (a team can have specific bounds for day affectation: for instance, a team should be affected at least three times to the Saturday day off). Shift affectations are modelled with similar constraint.



(a) A `flow` model (b) Experimentation results

Fig. 2. An employee scheduling problem

7.3 Bench results

This bench has been implemented in several versions and tested on at least ten runs: an `occurrence` model without explanations using a classical arc-consistency and backtracking algorithm in Java version of `Choco` [16], a `flow`

model without explanations using the same algorithms and a `flow` model with explanations using the `mac-dbt` algorithm [13] and `Palm`[12].

The results on Figure 2(b) show that the `flow` model is much more stable than the `occurrence` one. Indeed, as long as the problem is simple enough, the poor propagation of the `occurrence` constraint model is sufficient to find a solution. Furthermore, since propagation is really quick, found results are quite good compared to the `flow` model. But as soon as the problem becomes difficult enough (due to some conflicts between different `occurrence` constraints modelling subparts of the problem), this model is not efficient anymore: it needs a longer time to repair past decisions when a contradiction occurs.

Results show that explained version of the model is almost as efficient as the classical `model` whereas explanations need to be computed and maintained. Such efficiency is mainly due to precise explanations avoiding thrashing during searching a solution *and* a decrementality algorithm avoiding to compute a new support from scratch after each contradiction. Some further experimentations will be led so as to check if such a property is still true with complex search algorithms (with custom branchings for instance).

7.4 Explanations and user interaction

This application is supposed to be used by a final user. It means that when no solution is found, the user needs as much information as possible. Using explanations with the `flow` constraints provides two kinds of information: First, an explanation provides explicit inconsistent set of constraints: it is a precious information for the final user to modify the requests or to localise where the problem comes from, even if explanations must be modified to make them user-friendly [15]. Then, since the `flow` constraint is a global one involving numerous variables, the user may be interesting by a inconsistent subpart of the network; if this constraint is the only one responsible for a contradiction, instead of giving an explanation as introduced in definition 1, it can directly indicates variable bounds implying such a contradiction so as to point out bounds to modify.

8 Conclusion

In this paper, we introduced important points to be addressed when considering introducing explanations within global constraints. We illustrated our proposal with a flow constraint and provided experimental evaluation of our approach. To generate meaningful explanations, a theoretical study must be led to provide explanations as precise as possible, while ensuring a certain efficiency of the algorithm for practical purposes.

Our approach is particularly well designed for real complex problems as the bench showed. Further experimentations will be led to test this approach on big instances with real search algorithms.

Last, we already successfully instrumentated other global constraints (namely `stretch`, `alldifferent`, `gcc`) following the guidelines presented in this paper. More generic patterns could be covered: cyclic graphs or networks with associated cost to each edge. Such an explained global constraint menagerie may be useful for many optimisation problems.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, New York, 1993.
2. Nicolas Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
3. Claude Berge. *Graphes*. Gauthier-Villars, troisime edition, 1983.
4. Christian Bessière and Pascal Van Hentenryck. To be or not to be... a global constraint. In *CP'03*, 2003.
5. Alexander Bockmayr, Nicolai Pisaruk, and Abderrahmane Aggoun. Network flow problems in constraint programming. In *CP'01*, pages 196–210, 2001.
6. Romuald Debruyne, Gérard Ferrand, Narendra Jussien, Willy Lesaint, Samir Ouis, and Alexandre Tessier. Correctness of constraint retraction algorithms. In *FLAIRS'03*. AAAI press, 2003.
7. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
8. Matthew Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
9. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
10. Ulrich Junker. QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, 2001.
11. Narendra Jussien. The versatility of using explanations within constraint programming. Research Report 03/4/INFO, École des Mines de Nantes, France, 2003.
12. Narendra Jussien and Vincent Barichard. The palm system: explanation-based constraint programming. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, pages 118–133, Singapore, 2000.
13. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *CP'00*, pages 249–261, 2000.
14. Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139, July 2002.
15. Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP'01 11th Workshop on Logic Programming Environments (WLPE'01)*, Paphos, Cyprus, 1 December 2001.
16. François Laburthe. Choco: implementing a cp kernel. In *Proceedings of TRICS, a post-conference workshop of CP 2000*, Singapore, 2000.
17. Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 94, Twelth National Conference on AI*, pages 362–367, Seattle, 1994.
18. Jean-Charles Régin. Global constraints. Fourth international workshop CA-AI-OR, 2002.
19. Guillaume Rochart and Narendra Jussien. Explanations for a flow constraint (in French). Technical report, École des Mines de Nantes and Bouygues SA, 2003.
20. Guillaume Rochart, Narendra Jussien, and François Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, 2003.
21. Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problem. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
22. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
23. Gérard Verfaillie and Narendra Jussien. Dynamic constraint solving. In *CP'03*, 2003. Tutorial - online notes available http://www.emn.fr/jussien/CP03tutorial.

Arc Consistency in the Dual Encoding of Non-Binary CSPs

Panagiotis Karagiannis[1], Nikos Samaras[1], and Kostas Stergiou[2]

[1] Department of Applied Informatics
University of Macedonia - {pkar,samaras}@uom.gr
[2] Department of Information and Communication Systems Engineering
University of the Aegean - konsterg@aegean.gr

Abstract. A non-binary Constraint Satisfaction Problem (CSP) can be solved by converting the problem into an equivalent binary one and applying well-established binary CSP techniques. An alternative way is to use extended versions of binary techniques directly in the non-binary problem. There are two well-known methods in the literature for translating a non-binary CSP to an equivalent binary one; the hidden variable encoding and the dual encoding. It has been shown that arc consistency can be applied in the hidden variable encoding of a non-binary CSP with the same worst-case time complexity as generalized arc consistency in the non-binary representation. However, arc consistency in the dual encoding is so far considered prohibitively expensive to apply. In this paper we describe an arc consistency algorithm for the dual encoding with $O(e^3d^k)$ worst-case complexity. This gives an $O(d^k/e)$ saving compared to a generic algorithm and is close to the complexity of generalized arc consistency in the non-binary representation. Experimental results show that the new algorithm can be orders of magnitude better than an optimal generic arc consistency algorithm. Also, due to the stronger filtering achieved in the dual encoding compared to the non-binary representation, utilization of the new algorithm can make the dual encoding a competitive option for certain classes of non-binary constraints.

1 Introduction

Many problems from the real world can be represented as CSPs. For example, problems from timetabling, scheduling, resource allocation, planning, circuit design etc. Most of these problems can be naturally modeled using *n-ary* (or non-binary constraints). Because of this, an important part of research in constraint programming is focused on devising efficient filtering algorithms for specific non-binary constraints. Along these lines, efficient arc consistency algorithms are of primary importance. However, if no specialized filtering algorithm exists for some non-binary constraint then to process it we must either use a weak consistency algorithm (e.g. forward checking during search), or a generic but expensive algorithm for a stronger level of consistency (e.g. arc consistency). A third alternative is to translate the non-binary constraint into binary, and use well-developed techniques for binary constraints.

There are two widely known binary translations; the Dual Encoding (DE) [7] and the Hidden Variable Encoding (HVE) [11]. Recently, the efficiency of the above encodings has been studied theoretically and empirically [1, 14, 9, 2]. Among other theoretical results, it has been proved that Arc Consistency (AC) in the HVE achieves exactly the same consistency level as Generalized Arc Consistency (GAC) in the non-binary CSP [14]. Also, AC in the DE achieves a stronger level of consistency than GAC in the non-binary problem [14]. Experiments have shown that algorithms, such as maintaining arc consistency and forward checking, applied in the HVE can be competitive, and is some cases better, than their counterparts applied in the non-binary representation [1, 9]. For the DE, where all variables usually have large domains, AC is generally considered too expensive to apply. However, [13] has showed that, despite this disadvantage, there are problems where search in the dual encoding can be very effective.

In this paper we describe an AC algorithm for the DE of an arbitrary non-binary CSP with $O(e^3d^k)$ worst-case time complexity, where e denotes the number of constraints, d the maximum domain size of the variables, and k the maximum arity of the constraints. This is significantly lower compared to the $O(e^2d^{2k})$ complexity of a generic optimal AC algorithm. Also the complexity of the proposed algorithm is close to the $O(ekd^k)$ worst-case complexity of an optimal GAC algorithm for the non-binary representation. To achieve the $O(e^3d^k)$ bound, we take advantage of the micro-structure in the binary constraints of the DE by observing that these constraints are piecewise functional.

Experimental results with randomly generated problems and benchmark crossword puzzle generation problems demonstrate that applying AC using the proposed algorithm can offer a very significant speed-up (up to more than 2 orders of magnitude) compared to a generic AC algorithm (AC-2001 [6]). This improvement allows AC in the DE to be computationally feasible, when the constraints are tight. As a result, applying AC in the DE can be competitive and in some cases more effective that GAC in the non-binary representation and AC in the HVE, considering that it achieves a stronger level of consistency. This is demonstrated through experimental results with large sparse non-binary CSPs and crossword puzzles. We show that for large domain sizes the new algorithm is not only orders of magnitude faster than AC-2001, but can also be competitive with an optimal GAC algorithm for the non-binary representation.

The paper is organized as follows. In Section 2 we give some necessary background. In Section 3 we describe the new AC algorithm for the dual encoding. In Section 4 we present experimental results that demonstrate the practical value of the new algorithm. Finally, in Section 5 we conclude and discuss future work.

2 Preliminaries

A CSP is stated as a triple (X, D, C), where X is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is the domain of each variable $x_i \in X$, and C is a set of c constraints. Each *k-ary* constraint c is defined over a set of variables $(x_1, \ldots, x_k)$ by the subset of the *Cartesian* product $D(x_1) \times \ldots \times D(x_k)$ which are consistent

tuples. In this way, each constraint c implies the allowed combination of values for the variables $(x_1, \ldots, x_k)$. The verification procedure whether a given tuple is allowed by c or not is called a consistency check. An assignment of a value a to variable $x_i, i = 1, 2, \ldots, n$ is denoted by (x_i, a).

CSPs are usually represented as graphs, where nodes correspond to variables and edges to constraints. A solution of a CSP $G = (X, D, C)$ is an instantiation of the variables $x \in X$, such that all the constraints $c \in C$ are satisfied. A value $a \in D(x_i)$ is consistent with a binary constraint c_{ij} iff $\exists\ b \in D(x_j)$ such that the assignments (x_i, a) ,(x_j, b) are allowed by c_{ij}. In this case the value b is called a *support* for (x_i, a) on the constraint c_{ij}.

A value $a \in D(x_i)$ is *arc consistent* (AC) iff it has support in all $D(x_j)$ such that constraint $c_{ij} \in C$. A constraint c_{ij} is arc consistent if $\forall\ a \in D(x_i)$, $\exists\ b \in D(x_j)$ such that b is a support for (x_i, a). A binary CSP G is arc consistent iff all the constraints are arc consistent. The above definitions have been extended to non-binary CSPs. A non-binary constraint is GAC iff for any variable in the constraint and and any value of that variable, there exist compatible values for all the other variables in the constraint.

In what follows we will use the notation e for the number of constraints in a non-binary problem, d for the domain size of the original variables in the non-binary problem, and k for the arity of the non-binary constraints. For ease of analysis, the results presented are based on the assumption that all original variables have uniform domain size and all constraints are of the same arity. However, these restrictions can be easily lifted.

2.1 Dual Encoding

In the dual encoding each constraint c_i of the original n-ary CSP is represented by a dual variable v_i [7]. The domain of each dual variable consists of the set of allowed tuples in the original constraint. Binary constraints between two dual variables v_1 and v_2 exist iff the original constraints c_1 and c_2 share one or more variables. These binary constraints disallow pairs of tuples in which shared variables have different values. In order to show more clearly how the DE works, consider the following example with six original variables $x_i, i = 1, \ldots, 6$ with $0 - 1$ domains and four constraints $c_j, j = 1, \ldots, 4$. The four constraints of the CSP are:

$$x_1 + x_2 + x_6 = 1$$
$$x_1 - x_3 + x_4 = 1$$
$$x_4 + x_5 - x_6 \geq 1$$
$$x_2 + x_5 - x_6 = 0$$

Figure 1 depicts the DE for this problem. In the DE we have four dual variables $v_i, i = 1, 2, 3, 4$. The domains of these variables are the tuples that satisfy the respective constraint. For example, the dual variable v_2 associated with the second constraint has the domain $\{(0, 0, 1), (1, 0, 0), (1, 1, 1)\}$. There are binary compatibility constraints between dual variables whose corresponding non-binary constraints share one or more original variables. For example there is a constraint

between v_1 and v_2 (denoted by R11) because the corresponding non-binary constraints of v_1 and v_2 share original variable x_1. This constraint allows a pair of tuples $< t_i, t_j >$ iff the value of x_1 is the same in both t_i and t_j. For constraint R11 the allowed pairs of tuples are: $\{(< 0,0,1 >, < 0,0,1 >), (< 0,1,0 >, < 0,0,1 >), (< 1,0,0 >, < 1,0,0 >), (< 1,0,0 >, < 1,1,1 >)$.

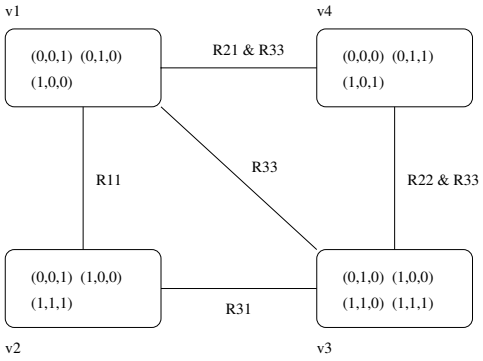


Fig. 1. Dual encoding of a non-binary CSP.

In the following, we will denote by c_{v_i} the non-binary constraint that is encoded by dual variable v_i. For an original variable x_j involved in c_{v_i}, $pos(x_j, c_{v_i})$ will denote the position of x_j in c_{v_i}. For instance, given a constraint c_{v_i} on variables x_1, x_2, x_3, $pos(x_2, c_{v_i}) = 2$.

3 Arc Consistency in the Dual Encoding

AC can be enforced in a binary CSP with O(ed^2) optimal worst-case time complexity. Since the DE is a binary CSP, one obvious way to apply AC is using a standard AC algorithm. The domain size of a dual variable corresponding to a $k-$ary constraint is of size d^k in the worst case. Therefore, if we apply an optimal AC algorithm (e.g. AC-2001 [6]) then we can enforce AC on one dual constraint with $O(d^{2k})$ worst-case complexity. In the DE of a CSP with e constraints of arity k there are at most $e(e-1)/2$ binary constraints (when all pairs of dual variables share one or more original variables). Therefore, we can enforce AC in the DE of the $k-$ary CSP with $O(e^2 d^{2k})$ worst-case complexity. This is significantly more expensive compared to the O(ekd^k) complexity bound of GAC in the non-binary representation and AC in the HVE[3]. Because of the very high complexity bound, AC processing in the DE is considered to be impractical, except perhaps for very tight constraints.

[3] AC in the HVE can applied with the same worst-case time complexity as GAC in the non-binary representation [9].

However, we will now show that AC can be applied in the DE much more efficiently using a simple algorithm. This algorithm can enforce AC on the DE of a n-ary CSP with $O(e^3d^k)$ worst-case time complexity. The improvement in the asymptotic complexity is based on the the observation that the constraints in the DE are piecewise functional. A constraint c on variables x_i and x_j is called *piecewise* if the domains of x_i and x_j can be partitioned into groups such that the elements of each group behave similarly with respect to c [8]. A *piecewise functional* constraint c on variables x_i and x_j is a constraint where the domains of x_i and x_j can be decomposed into groups such that each group of $D(x_i)$ (resp. $D(x_j)$) is supported by all values in at most one group of $D(x_j)$ (resp. $D(x_i)$) [8]. For example, the modulo (x_i MOD x_j) and integer division (x_i DIV x_j) constraints are piecewise functional. Based on [8], we can implement a simple algorithm that drastically reduces the complexity of AC filtering in the DE.

Consider a binary constraint between dual variables v_i and v_j. We can partition the tuples in the domain of either dual variable into groups such that all tuples in a group are supported by the same group of tuples in the other variable. If the non-binary constraints corresponding to the two dual variables share f original variables $x_1, \ldots, x_f$ of domain size d, then we can partition the tuples of v_i and v_j into d^f groups. Each tuple in a group s includes the same sub-tuple of the form $< a_1, \ldots, a_f >$, where $a_1 \in D(x_1), \ldots, a_f \in D(x_f)$. Each tuple τ in s will be supported by all tuples in a group s' of the other variable, where each tuple in s' also includes the sub-tuple $< a_1, \ldots, a_f >$.The tuples belonging to s' will be the only supports of tuple τ since any other tuple does not contain the sub-tuple $< a_1, \ldots, a_f >$. In other words, a group of tuples s in variable v_i will only be supported by a corresponding group s' in variable v_j where the tuples in both groups have the same values for the original variables that are common to the two encoded non-binary constraints. This partitioning of the dual variable domains means that the constraints in the DE are piecewise functional.

Example 1. Assume we have two dual variables v_1 and v_2. v_1 encodes constraint (x_1, x_2, x_3), and v_2 encodes constraint (x_1, x_4, x_5), where the original variables $x_1, \ldots, x_5$ have the domain $\{0, 1, 2\}$. We can partition the tuples in each dual variable into 3 groups. The first group will include tuples of the form $< 0, *, * >$, the second will include tuples of the form $< 1, *, * >$, and the third will include tuples of the form $< 2, *, * >$. A star (*) means that the corresponding original variable can take any value. Each group is supported only by the corresponding group in the other variable. Note that the tuples of a variable v_i are partitioned in different groups according to each constraint that involves v_i. For instance, if there is another dual variable v_3 encoding constraint (x_6, x_7, x_3) then the partition of tuples in v_1 according to the constraint between v_1 and v_3 is into groups of the form $< *, *, 0 >$, $< *, *, 1 >$, $< *, *, 2 >$.

In Figure 2 we sketch an AC-3 like AC algorithm for the DE, which we call PW-AC (*PieceWise Arc Consistency*). As do most AC algorithms, PW-AC uses a stack to propagate deletions from the domains of variables. For reasons that we will explain bellow, this stack processes groups of piecewise partitions, instead

```
procedure PW − AC
1:  Q ← ∅
2:  initialize all group counters to 0
3:  for each variable v_i
4:      for each variable v_j constrained with v_i
5:          for each tuple τ ∈ D(v_i)
6:                  counter(GroupOf(S(v_i, v_j), τ)) ← counter(GroupOf(S(v_i, v_j), τ)) + 1
7:  for each variable v_i
8:      for each variable v_j constrained with v_i
9:          for each group s_l(v_i, v_j)
10:             if counter(s_l(v_i, v_j)) = 0
11:                 put s_l(v_i, v_j) in Q
12: return Propagation

function Propagation
13: while Q is not empty
14:     pick group s_l(v_i, v_j) from Q
15:         δ ← ∅
16:         δ ← Revise(v_i, v_j, s_l(v_i, v_j))
17:             if D(v_j) is empty return INCONSISTENCY
18:             for each group s_l'(v_j, v_k) in δ put s_l'(v_j, v_k) in Q
19: return CONSISTENCY

function Revise(v_i, v_j, s_l(v_i, v_j))
20: for each tuple τ ∈ D(v_j) where τ ∈ sup(s_l(v_i, v_j))
21:     remove τ from D(v_j)
22:     for each group s_l'(v_j, v_k) that includes τ
23:         counter(s_l'(v_j, v_k)) ← counter(s_l'(v_j, v_k)) − 1
24:         if counter(s_l'(v_j, v_k)) = 0
25:             add s_l'(v_j, v_k) to δ
26: return δ
```

Fig. 2. PW-AC. An AC algorithm for the dual encoding.

of variables or constraints as is usual in AC algorithms. We use the following notation:

- $S(v_i, v_j) = \{s_1(v_i, v_j), \ldots, s_m(v_i, v_j)\}$ denotes the piecewise partition of $D(v_i)$ with respect to the constraint between v_i and v_j. Each $s_l(v_i, v_j)$ is a group of the partition.
- $sup(s_l(v_i, v_j))$ denotes the group of $S(v_j, v_i)$ that can support group $s_l(v_i, v_j)$ of $S(v_i, v_j)$. As discussed, this group is unique.
- $counter(s_l(v_i, v_j))$ holds the number of tuples that belong to group $s_l(v_i, v_j)$ of partition $S(v_i, v_j)$ and are present in $D(v_i)$. That is, at any time the value of $counter(s_l(v_i, v_j))$ gives the current cardinality of the group.
- $GroupOf(S(v_i, v_j), \tau)$ is a function that returns the group of $S(v_i, v_j)$ where tuple τ belongs. To implement this function, for each constraint between dual variables v_i and v_j we store the original variables shared by the non-binary constraints c_{v_i} and c_{v_j}. Also, for each such original variable x_l we store $pos(x_l, c_{v_i})$ and $pos(x_l, c_{v_j})$. In this way the $GroupOf$ function takes constant time.
- The set δ contains the groups that have their counter reduced to 0 after a call to function *Revise*. That is, groups such that all tuples belonging to them have been deleted.

The algorithm works as follows. In an initialization phase, for each group we count the number of tuples it contains (lines 3–6). Then, for each variable v_i we iterate over the variables v_j that are constrained with v_i. For each group $s_l(v_i, v_j)$ of $S(v_i, v_j)$, we check if $s_l(v_i, v_j)$ is empty or not (line 9). If it is empty, it is added to the stack for propagation.

In the next phase, function *Propagation* is called to propagate the deletions (line 12). Once the previous phase has finished, the stack will contain a number of groups with 0 cardinality. For each such group $s_l(v_i, v_j)$ we must remove all tuples belonging to group $sup(s_l(v_i, v_j))$ since they have lost their support. This is done by successively removing a group $s_l(v_i, v_j)$ from the stack and calling function *Revise*. Since group $sup(s_l(v_i, v_j))$ has lost its support, each tuple $\tau \in D(x_j)$ that belongs to $sup(s_l(v_i, v_j))$, is deleted (line 21). Apart from $sup(s_l(v_i, v_j))$, tuple τ may also belong to other groups that $D(v_j)$ is partitioned in with respect to constraints between v_j and other variables. Since τ is deleted, the counters of these groups must be updated (i.e. reduced by one). This is done in lines 22–23. If the counter of such a group becomes 0 then the group is added to the stack for propagation (lines 24–25). The process stops when either the stack or the domain of a variable becomes empty. In the former case, the DE is AC, while in the latter it is not.

3.1 Time Complexity

The PW-AC algorithm consists of two phases. In the initialization phase we set up the group counters, and in the main phase we delete unsupported tuples and propagate the deletions. We now analyze the time complexity of PW-AC.

Theorem 1. *The worst-case time complexity of algorithm PW-AC is* $O(e^3d^k)$.

Proof. We assume that for any constraint in the dual encoding, the non-binary constraints corresponding to the two dual variables v_i and v_j share at most f original variables $x_1, \ldots, x_f$ of domain size d. This means that each partition consists of at most d^f groups. Obviously, f is equal to $k-1$, where k is the maximum arity of the constraints. In the initialization phase of lines 3–6 we iterate over all constraints, and for each constraint between variables v_i and v_j, we iterate over all the tuples in $D(v_i)$. This gives $\mathrm{O}(e^2d^k)$ asymptotic time complexity. Thereafter, function *Revise* is called only when a group becomes empty. This means that it is called at most d^f times for each constraint. Each time *Revise* is called for a group $s_l(v_i, v_j)$, we iterate over the (at most) d^{k-f} tuples of group $sup(s_l(v_i, v_j))$, and for each tuple τ we update the counters of the groups where τ belongs. There are at most e such groups (in case v_j is constrained with all other dual variables). Therefore, for one constraint the main phase of the algorithm costs $\mathrm{O}(d^f d^{k-f} e)$=$\mathrm{O}(ed^k)$ operations. For the e^2 constraints of the dual encoding, the complexity, including the preprocessing step, is $\mathrm{O}(e^2d^k + e^3d^k)$=$\mathrm{O}(e^3d^k)$. *Q.E.D*

3.2 Space Complexity

Since we deal with extensionally specified constraints, the asymptotic space complexity of PW-AC, and any GAC (or AC) algorithm on the non-binary representation (or binary encoding), is dominated by the $\mathrm{O}(ed^k)$ space need to store the allowed tuples of the non-binary constraints. Algorithm PW-AC also requires $\mathrm{O}(e^2d^f)$ space to store the counters for all the groups, $\mathrm{O}(e^2d^f)$ space for the stack, and $\mathrm{O}(fe^2)$ space for the fast implementation of function $GroupOf$.

4 Experimental Study

To evaluate the performance of algorithm PW-AC, we run some experiments on randomly generated problems and benchmark crossword puzzles. First, we compared the run times of PW-AC and AC-2001 in the dual encoding of random ternary CSPs. Then we compared the run times of the algorithms in the DE to the run time of GAC-2001 on the non-binary representation, and we also measured the number of times AC in the dual encoding was able to determine insolubility compared to GAC in the non-binary representation. Random instances were generated using the extended *model B* as it is described in [5]. To summarize this generation method, a random non-binary CSP is defined by the following five input parameters:

n - number of variables
d - domain size
k - arity of the constraints
p - density percentage of the generated graph
q - looseness percentage of the constraints

In the following, a class of non-binary CSPs will be denoted by a tuple of the form $< n, d, p, q >$. All constraints are ternary. We use a star (*) for the case where one of the parameters is varied. For example, the tuple $< 50, 20, 0.1, * >$ stands for the class of problems with 50 variables, domain size 20, graph density 0.1, and varying constraint looseness. For each set of parameters we generated 100 instances and measured the average run times. All the cpu times reported are in milliseconds. Initialization times for the data structures of PW-AC are included in the results.

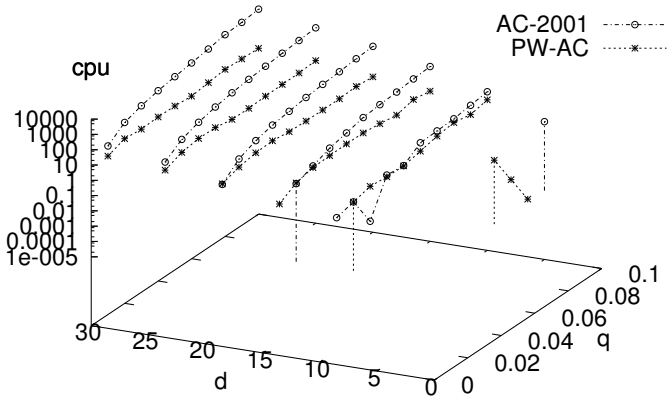


Fig. 3. Cpu times in $< 15, *, 0.04, * >$ problems.

The first set of experiments was carried out on small problems with 15 and 30 variables. We present part of the results to give a flavor of the relative performance of AC-2001 and PW-AC under varying domain size, graph density, and constraint looseness. Figures 3, 4, 5 compare the performance of AC-2001 and PW-AC in the DE of problems with 15 variables. Experiments were carried out on a class of sparse problems (Figure 3), a class of medium density (4), and a class of dense problems (5). In all figures we vary the domain size in steps of 5, starting from 5 up to 30. The looseness of the constraints is varied in steps of 0.01, starting from 0.01 up to 0.1. As we can see, AC-2001 is faster in problems with small domain sizes. However, these problems are very easy and the differences are not important. PW-AC starts to outperform AC-2001 as the domain size grows, and the problems become harder. The difference which reaches more than one order of magnitude is more notable when the constraint looseness is higher. This is expected, since as the number of allowed tuples in a constraint grows, AC-2001 takes more and more time to find supports. In relation to the

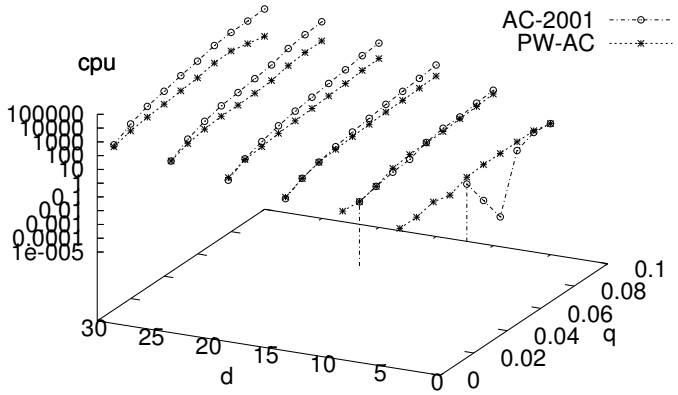


Fig. 4. Cpu times in $< 15, *, 0.09, * >$ problems.

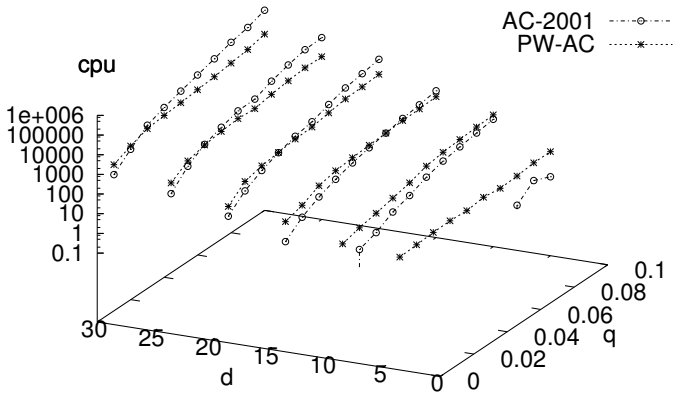


Fig. 5. Cpu times in $< 15, *, 0.3, * >$ problems.

constraint graph density, note that the difference between PW-AC and AC-2001 becomes smaller as the problems become tighter. This is caused, partly by the higher initialization times of PW-AC (since the number of group counters grows), and mainly by the higher number of group counter updates that PW-AC performs (lines 22–23 of the algorithm). Note that for dense CSPs, GAC-2001 in the non-binary representation is much faster than both AC algorithms in the DE, and it is unlikely that the DE will pay off, despite the higher consistency level that it achieves.

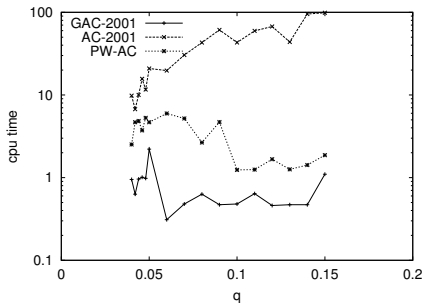


Fig. 6. Cpu times in $< 50, 10, 0.003, * >$ problems.

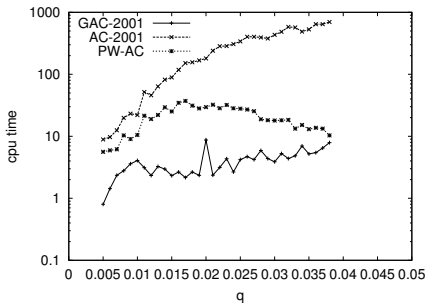


Fig. 7. Cpu times in $< 50, 20, 0.003, * >$ problems.

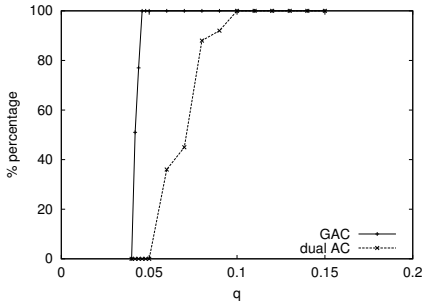


Fig. 8. Percentage of GAC and AC in the DE instances in $< 50, 10, 0.003, * >$ problems.

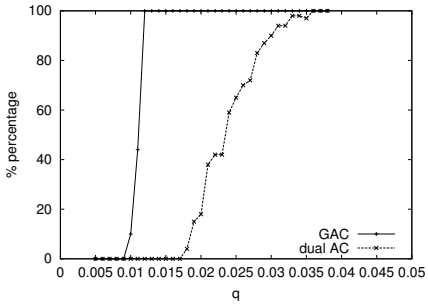


Fig. 9. Percentage of GAC and AC in the DE instances in $< 50, 20, 0.003, * >$ problems.

Having established that PW-AC is significantly more efficient than AC-2001, we investigated whether the savings achieved can make the DE competitive with the non-binary representation. From our experiments we conjecture that this is the case for sparse problems, while for CSPs with medium and high density the non-binary representation is preferable. To demonstrate the point about sparse problems, Figure 6 gives average cpu times of GAC-2001 in the non-binary representation, and AC-2001, PW-AC in the DE for problems with

50 variables, domain size 10, and 0.003 graph density (58 3-ary constraints). Figure 8 shows the percentage of instances that are GAC in the non-binary representation and AC in the DE for the same class of problems. Similar plots are repeated in Figures 7, 10, 11 where we give cpu times of GAC-2001, AC-2001 and PW-AC for the same class of problems, in terms of number of variables and graph density, and domain size 20, 30, and 40. Figures 9, 12, 13 give the corresponding percentages of GAC and AC instances.

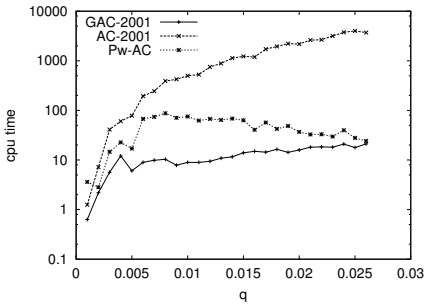


Fig. 10. Cpu times in $< 50, 30, 0.003, * >$ problems.

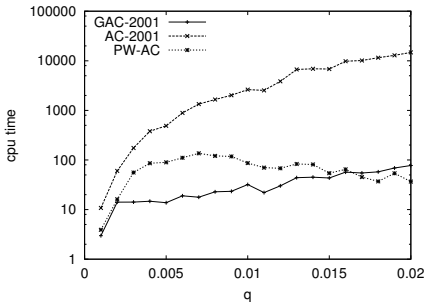


Fig. 11. Cpu times in $< 50, 40, 0.003, * >$ problems.

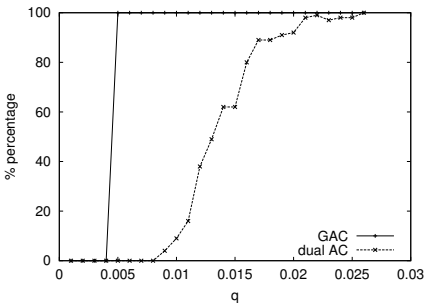


Fig. 12. Percentage of GAC and AC in the DE instances in $< 50, 30, 0.003, * >$ problems.

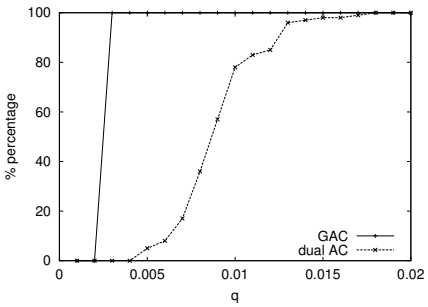


Fig. 13. Percentage of GAC and AC in the DE instances in $< 50, 40, 0.003, * >$ problems.

First, note that the difference between PW-AC and AC-2001 constantly rises as the looseness becomes higher. At the AC phase transition region (i.e. at the point where around half instances are AC and the other half are not) the difference is one order of magnitude. Beyond that point the difference reaches up to three orders of magnitude.

Second, GAC-2001 is faster than PW-AC (up to one order of magnitude) when the looseness in low, but the difference becomes smaller as the looseness

grows. For constraints with many allowed tuples, PW-AC is actually faster than GAC-2001. Also, as the domain size is increased the difference in favor of GAC-2001 for tight constraints, becomes less notable. As we increase the domain size, the differences in favor of PW-AC for looser constraints become more remarkable. For example, in problems with 50 variables, domain size 100, graph density 0.003 and constraint looseness 0.005, the average run times of GAC-2001, AC-2001, and PW-AC in msecs were 2250, 333641, and 78, respectively.

Third, from the figures showing percentages of GAC and AC we can see that AC in the DE determines insolubility for problems in a large range of constraint looseness that are GAC. Also, we noted that in many instances that are both GAC and AC in the DE, the AC algorithms in the DE pruned a significantly greater number of values than GAC in the non-binary representation. These are effects of the higher consistency level achieved by AC in the DE. If we use algorithm AC-2001 to apply AC in the DE we cannot exploit this advantage of the DE because of the very high run times. However, using algorithm PW-AC we can exploit the higher consistency level achieved in the DE by paying a run time penalty for tight constraints, and actually winning in both consistency level and run times for looser constraints. We repeat that this is the case only for sparse problems.

4.1 Maintaining Arc Consistency

We also evaluated the effect that algorithm PW-AC has when maintaining arc consistency during search (MAC algorithm [12]). We compared three algorithms; MAC in the DE that maintains AC using AC-2001 (MAC-2001), MAC in the DE that maintains AC using PW-AC (MAC-PW-AC), and MAC in the non-binary representation that maintains MGAC using GAC-2001 (MGAC). For this comparison we used benchmark crossword puzzle generation problems.

Crossword puzzle generation problems have been used before for the evaluation of binary encodings of non-binary problems [1, 14]. In these problems we try to construct puzzles for a given number of words and a given grid which has to be filled in with words. This problem can be represented as either a non-binary or a binary CSP in a straightforward way. In the non-binary representation there is a variable for each letter to be filled in and a non-binary constraint for each set of k variables that form a word in the puzzle. The domain of each variable consists of the low case letters in the English alphabet giving a domain size of 26. The allowed tuples of such a constraint are all the words with k letters in the dictionary used. These are very few compared to the 26^k possible combinations of letters, which means that the constraints are very tight. In the dual encoding there is a variable for each word of length k in the puzzle and the possible values of such a variable are all the words with k letters in the dictionary. This gives variables with large domains (up to 4072 values for the Unix dictionary that we use in the experiments). There are binary constraints between variables that intersect (i.e. they have a common letter).

Table 1 compares the cpu times of the two MAC algorithms in the DE and MGAC in the non-binary representation on various crossword puzzles taken from [3]. All algorithms use the dom/deg heuristic for variable ordering [4].

puzzle	n	m	MGAC	MAC-2001	MAC-PW-AC
15.01	78	189	1.32 (574)	69.67 (770)	4.43
15.02	80	191	3.70 (1312)	154.82 (1822)	13.15
15.03	78	189	0.64 (338)	21.70 (128)	1.06
15.04*	76	193	39.65 (19677)	—	—
15.05	78	181	0.46 (286)	14.56 (195)	1.20
15.07	74	193	95.12 (12733)	—	—
15.08	84	186	0.43 (247)	20.23 (222)	1.57
15.09	82	187	0.40 (251)	—	—
19.05**	126	291	0.04 (0)	9.84 (0)	0.04
19.06	128	287	4.07 (375)	23.78 (171)	4.03
19.07	134	291	4.18 (305)	19.34 (152)	4.08
19.08	134	291	—	38.51 (610)	12.09
19.09	130	295	1.45 (308)	21.45 (192)	3.81
19.10**	128	291	0.04 (0)	9.71 (0)	0.04
21.02	130	295	2.14 (637)	76.92 (452)	10.81
21.03	130	295	351.93 (78146)	269.13 (3012)	35.76
puzzle20	80	187	0.4 (216)	15.54 (122)	0.91
6×6	12	36	8.92 (2263)	95.46 (2198)	5.93
9×9*	18	81	49.71 (4972)	3382.82 (9580)	117.48
10×10*	20	100	11.81 (1027)	474.54 (1409)	22.78

Table 1. Comparison (in cpu time) between MAC algorithms for the DE and MGAC for the non-binary representation of crossword puzzles. In brackets we give the number of node visits (MAC-2001 and MAC-PW-AC visit the same nodes). **n** is the number of words and **m** is the number of blanks. An em-dash (—) is placed wherever the algorithm did not manage to find a solution within 2 hours of cpu time. Problems marked by (*) are insoluble. Problems marked by (**) are arc inconsistent.

From the data in Table 1 we can clearly see that MAC-PW-AC is significantly faster than MAC-2001 on all instances. The speedup offered by the use of PW-AC makes MAC in the DE competitive with MGAC in many cases where there was a clear advantage in favor of MGAC. Also, in some instances (e.g. puzzle 21.03), the use of PW-AC makes MAC in the DE considerably faster than MGAC. However, there are still some instances where MGAC finds a solution (or proves insolubility) fast, while MAC in the DE thrashes.

5 Conclusion

In this paper we described an AC algorithm for the dual encoding of an arbitrary non-binary CSP with $O(e^3d^k)$ worst-case time complexity. This offers a

significant improvement compared to the $O(e^2d^{2k})$ complexity of a generic optimal AC algorithm. Also the complexity of the proposed algorithm is close to the $O(ekd^k)$ worst-case complexity of an optimal GAC algorithm for the non-binary representation. The $O(e^3d^k)$ bound is achieved by taking advantage of the fact that constraints in the dual encoding are piecewise functional. Experimental results with randomly generated problems and benchmark crossword puzzles demonstrated that applying AC using the proposed algorithm can offer a very significant speed-up compared to a generic AC algorithm. As a result, applying AC in the dual encoding can be competitive and potentially more effective that GAC in the non-binary representation and AC in the hidden variable encoding, considering that it achieves a stronger level of consistency. This opens some interesting directions for future work in non-binary CSP solving. We intend to investigate the possibility of using the dual encoding to propagate non-binary constraints to take advantage of stronger filtering, while instantiating the original variables during search to take advantage of variable ordering heuristics.

References

1. F. Bacchus and P. Van Beek. On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems. In *Proceedings of AAAI'98*, pages 310–318.
2. F. Bacchus, X. Chen, P. van Beek and T. Walsh. Binary vs. Non-Binary CSPs. Artificial Intelligence, 2002.
3. A. Beacham, X. Chen, J. Sillito and P. Van Beek. Constraint Programming Lessons Learned from Crossword Puzzles. In *Proceedings of the 14th Canadian Conference in AI'2001,Canada*.
4. C. Bessière and J.C. Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of CP'96*, pages 61–75, 1996.
5. C. Bessière, P. Meseguer, E. C. Freuder and J. Larrosa. On Forward Checking for Non-binary Constraint Satisfaction. In *Proceedings of CP'99*, pages 88–102.
6. C. Bessière and J. C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'2001*.
7. R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
8. P. Van Hentenryck, Y. Deville and C. Teng. A Generic Arc Consistency Algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
9. N. Mamoulis and K. Stergiou. Solving non-binary CSPs using the Hidden Variable Encoding. In *Proceedings of CP'2001*.
10. R. Mhor and G. Masini. Arc and Path Consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
11. F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI'90*, pages 550–556.
12. D. Sabin, and E. Freuder. Contradicting conventional wisdom inconstraint satisfaction. In *Proceedings of PPCP'94*, Seattle, WA, 1994.
13. B. Smith. A Dual Graph Translation of a Problem in 'Life'. In *Proceedings of CP-2002*, pages 402–414.
14. K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In *Proceedings of AAAI'99*, pages 163–168.

Arc Consistency in MAC: A New Perspective*

Chavalit Likitvivatanavong, Yuanlin Zhang,
James Bowen, and Eugene C. Freuder

Cork Constraint Computation Centre, University College Cork, Ireland
{chavalit, yzhang, j.bowen, e.freuder}@4c.ucc.ie

Abstract. AC refers to algorithms that enforce arc consistency on a constraint network while MAC refers to a backtracking search scheme where after each instantiation of a variable, arc consistency is maintained (or enforced) on the new network. In this paper, we use *mac* to denote *maintaining arc consistency* in MAC. In all existing studies, *mac* is simply taken as an associate of an AC algorithm. In this paper, we argue that it is worth taking *mac* as an entity separated from AC. Based on an observation that *mac* is invoked many times during a search, we propose three schemes to improve the efficiency of *mac*. First, the results of constraint checks are cached. Second, values remained in the auxiliary data structures used by sophisticated AC algorithms are better exploited. Third, we adopt a non-invariant ordering on domain values. Algorithms are also presented for these schemes. Their performances are discussed in terms of time complexity, space complexity, number of checks, and running time. In our experimental setting, we find that it is possible to design a *mac* algorithm which is simple to implement and runs faster as well as uses less space.

1 Introduction

AC refers to algorithms that enforce arc consistency on a constraint network while MAC refers to a backtracking search scheme where after each instantiation of a variable, arc consistency is maintained (or enforced) on the new network. MAC is regarded as one of the best search schemes not only by the researchers in the community, but also by the practitioners in the industry. It has been employed by many constraint solvers, for instance ILOG solver and CHOCO.

We will use *mac* to denote the *maintaining arc consistency* component of a MAC algorithm. It is taken for granted that to improve the efficiency of a MAC algorithm, one needs to focus on the AC algorithm employed by MAC; conversely, the effectiveness of AC algorithm is usually shown in the context of MAC. This reasoning works well and produces significant insight on the efficiency of AC algorithms. Unfortunately as a result, the larger context of MAC is completely ignored. In other words, AC is simply equated to *mac*. In this paper we argue that although *mac* is similar to AC, it deserves separate treatment.

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

A number of AC algorithms have been designed since Waltz first proposed a scene filtering algorithm in [9]. Fundamentally, they fall into two classes: coarse-grained (e.g. AC3 [5], AC3.1 [10] and AC2001 [2]) and fine-grained (e.g. AC6 [1]). When a value is removed from a domain, algorithms in the former class will revise any affected domain with respect to the corresponding constraint, whereas algorithms in the latter class will revise only the affected values. There are algorithms in both classes which have optimal worst-case time complexity and perform well in empirical studies. Sabin and Freuder [7] proposed MAC in 1994 and it becomes so well accepted in the last decade by the community that to show the efficiency of a new AC algorithm, one has to test it in MAC. It has been shown that most efficiency improvements of AC algorithms could also lead to the improvements of MAC.

The most obvious difference between AC and *mac* is that an AC algorithm is executed only once, but due to backtracking and failed assignment of a variable, a *mac* algorithm may be executed many times (tens of thousands of times in a difficult random problem instance with 150 constraints and 50 variables, each of which has a domain with 30 values). Taking this massive number of executions into account, we propose three improvements for *mac*.

The fist idea involves caching the results of constraint checks. Unlike AC, for *mac* the same constraint checks may be repeated *enormous* amount times (in one of our experiments, for a problem that can be solved in 10 to 20 minutes, *mac* needs billions of checks whereas AC uses only millions.) Therefore, not only is it sensible to memorize the results for future use, it is *essential* for hard problems or the problems in which the cost of constraint checks is high.

Indeed, the number of constraint checks alone cannot be used to determine precisely the empirical performance of an algorithms due to the uncertainty on the cost of a constraint check. By using cache, we are able to deal with this uncertainty better in *mac*, making the impact of the cost of constraint checks more controllable. In our experiments, the performance of *mac*3 is significantly improved by using cache, even under reasonably cheap constraint checks. More details will be presented in Section 3.

The second idea involves passing information from one execution of *mac* to another. Most AC algorithms uses auxiliary data structures to speed up the execution time in standalone preprocessing but they cannot be used effectively in *mac*. The main problem lies in the rigid invariant associated with this structure and the total ordering of variable domains, which require expensive maintenance when backtrack. In section 4, we will present a method that relax the invariant and better exploit this auxiliary data structures. This also lead to the third idea which involves adaptive domain ordering, to be presented in Section 5. Section 6 describes related works. The paper is concluded in Section 7.

These issues will be studied using coarse-grained algorithms, specifically AC3 and AC3.1. The reason lies in their simplicity, and in the case of AC3.1, its optimal worst-case time complexity. Both algorithm are also empirically efficient.

2 Preliminaries

In this section we review AC3, AC3.1, and MAC. We introduce *mac* and experimental settings used in next few sections.

Definition 1 (Binary Constraint Network) A *binary constraint network* is a triplet $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a finite set of variables, $\mathcal{D} = \{D_V \mid V \in \mathcal{V}\}$ where D_V is a finite set of possible values for V, and $\mathcal{C}$ is a finite set of constraints such that each $C_{XY} \in \mathcal{C}$ is a subset of $D_X \times D_Y$ indicating the compatible pairs of values for X and Y, where $X \neq Y$. Deciding whether $(a, b) \in C_{XY}$ is called a *constraint check*; this is sometimes written as a boolean function $C_{XY}(a, b)$. If $C_{XY} \in \mathcal{C}$, then $C_{YX} = \{(y, x) \mid (x, y) \in C_{XY}\}$ is also in $\mathcal{C}$.

For $(a, b) \in C_{XY}$, b is called a *support* of a in Y. If a value $a \in D_X$ has no support in Y where $C_{XY} \in \mathcal{C}$ then a is *invalid*. A constraint C_{XY} is *arc consistent* if every value $a \in D_X$ has a support in D_Y. A constraint network is *arc consistent* if all constraints are arc consistent. If a constraint network is not arc consistent, it can be made so by removing invalid values. Such an algorithm is called Arc Consistency (AC) algorithm.

Throughout this paper, we use n, e, and d to denote the number of variables, the number of constraints, and the maximum domain size in the network. We use D_V^0 to distinguish the *original domain* of V, while D_V denotes the *current domain*, which may change in the course of search. Deciding whether a value is in the current domain is called a *domain check*. A value in D_V^0 is *alive* if it is in D_V; otherwise it is said to be *pruned*.

Function $\texttt{succ}(b,\ D_Y)$ is a successor function defined in the usual sense. head and tail denote the start and the end of a domain, of which they are not members. succ(head) gives the first value while succ of the last value returns tail; if the domain is empty then succ(head) = tail. Function cirSucc ("circular successor") is defined as follows: $\texttt{cirSucc}(x) = \textsf{head}$ if $\texttt{succ}(x) = \textsf{tail}$; otherwise $\texttt{cirSucc}(x) = \texttt{succ}(x)$

2.1 AC3 and AC3.1

AC3 and AC3.1 are two representatives of coarse-grained algorithms. A basic operation a constraint *revision*, that is, removing invalid values in order to make the network arc consistent. The results are then *propagated* to other connecting variables, entailing more revisions. The generic algorithm AC() for coarse-grained algorithms is listed below. We also list AC3 and AC3.1 whose pseudo-code include only procedures different from the generic algorithm. In both algorithms, we assume a total ordering on each domain.

The difference between AC3 and AC3.1 lies in the routine hasSupport. AC3 finds a support from scratch while AC3.1 finds it by using the support found in the previous revision as a starting point. AC3.1 uses a data structure $\textsf{last}(X, a, Y)$ to remember the last support of a in D_Y, where $a \in D_X$.

AC

```
AC()
 1 ACInitialize()
 2 Q ← {(X, Y) | C_XY ∈ C}
 3 return propagate(Q)
propagate(Q)
 4 while Q ≠ ∅ do
 5 |   select and delete an arc (X, Y) from Q
 6 |   if revise(X, Y) then
 7 |   |   if D_X = ∅ then return failure
 8 |   |   Q ← Q ∪ {(W, X) | C_WX ∈ C, W ≠ Y}
 9 return success
revise(X, Y)
10 delete ← false
11 foreach a ∈ D_X do
12 |   if not hasSupport(X, a, Y) then
13 |   |   remove a from D_X
14 |   |   delete ← true
15 return delete
hasSupport(X, a, Y) {}
ACInitialize() {}
```

AC3

```
hasSupport(X, a, Y)
 1 b ← head
 2 while b ← succ(b, D_Y) and b ≠ tail do
 3 |   if C_XY(a, b) then return true
 4 return false
```

AC3.1

```
ACInitialize()
 1 foreach C_XY ∈ C and a ∈ D_X do last(X, a, Y) ← head
hasSupport(X, a, Y)
 2 b ← last(X, a, Y)
 3 if b ∈ D_Y then return true
 4 while b ← succ(b, D_Y) and b ≠ tail do
 5 |   if C_XY(a, b) then
 6 |   |   last(X, a, Y) ← b
 7 |   |   return true
 8 return false
```

The worst-case time complexity of AC3 is $O(ed^3)$ [5], while the worst-case complexity of AC3.1 is optimal at $O(ed^2)$. AC3 does not use any auxiliary data structure, whereas the space complexity of the auxiliary data structure of AC3.1 is $O(ed)$.

One way to evaluate the empirical performance of the AC algorithms is to count the number of constraint checks they conduct. However, an algorithm that has fewer number of constraint checks may consume more CPU time than another one with more checks. In our experiment, we also count the number of domain checks whose cost become more significant as the cost of a constraint check goes down.

2.2 MAC

MAC [7] is a backtracking search scheme (see Fig. 1) for finding a solution of a constraint network. Under this scheme, the network is preprocessed by an AC algorithm. During search, arc consistency is maintained (or enforced) after each instantiation of a variable in order to prune the search space. This process of maintaining arc consistency is denoted by *mac*; in the figure, V denotes the most recent assigned variable. Obviously, a key component of *mac* is AC, i.e. *mac* subsumes AC. In this paper, *mac* based on AC3 is called mac3, and *mac* based on AC3.1 called mac3.1. A generic pseudo-code for *mac* is listed below.

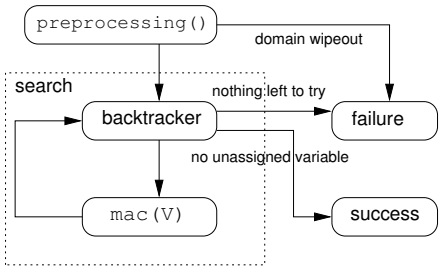


Fig. 1. MAC schema

The main difference between *mac* and AC is that AC is executed only once, but *mac* is executed whenever a variable is instantiated or when backtracking occurs. Given an AC algorithm, it is easy to design a *mac* algorithm based on it. For instance, mac3 is derived from AC3 and functions exactly like AC3. In the following sections we will add more functionalities that take advantage of the MAC environment.

mac

```
mac(V)
  1 macInitialize()
  2 Q ← {(U, V) | C_UV ∈ C}
  3 return propagate(Q)
macInitialize() {}
```

2.3 Repeated Constraint Checks

During search in MAC, some constraint checked may be repeated even though an individual run of *mac* is optimal. We identify the following types of repeated constraint check during search:

- **positive repeat** A constraint check between a and b is performed, and as a result b supports a. Later in the search, during which period a and b remain in their respective domains at all time, (a, b) is checked again.
- **negative repeat** A constraint check between a and b is performed, and as a result b does *not* support a. Later in the search, during which period a and b remain in their respective domains at all time, (a, b) is checked again.

2.4 Experimental Settings

We use a backtracking algorithm that dynamically picks a variable with minimum domain to be instantiated first. Since both AC and *mac* involve frequent operations on a variable domain, we design a domain as a *random-accessed doubly linked-list*. Under this implementation, *the following operations take constant time*: (1) deleting a value, (2) checking whether a value is in the current domain (3) given a value, returning the next value in the current domain.

In the experiments reported in this paper we compare a number of algorithms against AC3, so to be fair we try to reduce as much as possible the amount of time needed to perform constraint checks, which dominates the overall running time of AC3. To this end, we use explicit constraint storage in our implementation. However, given wide-ranging applications of CSPs, it is better to make use of a function call that determines whether a given tuple is allowed by a constraint. As a result, the total cost of a constraint check in our implementation is the overhead of a function call plus the cost for memory lookup.

We use random problems to compare the experimental performance of our proposals, based on model B [3]. It is parameterized as (n, d, e, tightness) where tightness is the number of tuples *disallowed* by a constraint. We control the difficulty of the generated instances by varying the tightness. Results are averaged over 10 different instances. Since we observed a large variance on easy problems, all statistic on easy problems are averaged over three batches of executions, each containing 10 instances.

The algorithms are written in C++ and compiled with g++. Running time of specific routines are profiled using gprof. The experimental platform is Linux

2.4.20 running on a Dell PowerEdge 4600 which has two Intel Xeon 2.80GHz CPU's and 4GB of RAM.

3 Caching Constraint Checks

Our proposal relies the fact that *mac* will be called many times and thus the consistency of the same tuple may be checked repeatedly. Given the uncertainty of the cost of constraint checks, it is worth recording the result so that the next time the same check is requested it will be answered with little cost.

For simplicity, we cache the result of every possible constraint check. In a binary constraint network, the size of the cache is $O(ed^2)$ because there are e constraints and for each constraint there are at most d^2 tuples.

An immediate consequence of this approach is that the performance of *mac* with cache would be significantly improved even if each constraint check is moderately expensive. Another benefit of this approach is that we can now assume the constraint check is reasonably cheap because the cost of the management of the cache and the cost of the initial $O(ed^2)$ raw constraint checks are amortized over a large amount of repeated checks.

The cache idea is tested by using AC3. The reason for this choice of algorithm is that the physical CPU time of AC3 is good but the number of its constraint checks is usually several times of those of more sophisticated algorithms. Using cache may benefit AC3 the most. The new algorithm is named mac3cache and listed below.

mac3cache

```
preprocessing()
  1 AC3()
ACInitialize()
  2 foreach C_XY ∈ C and a ∈ D_X and b ∈ D_Y do cache(C_XY, a, b) ← nil
hasSupport(X, a, Y)
  3 b ← head
  4 while b ← succ(b, D_Y) and b ≠ tail do
  5 |   if cache(C_XY, a, b) = nil then
  6 |     cache(C_XY, a, b) ← C_XY(a, b)
  7 |   if cache(C_XY, a, b) then return true
  8 return false
```

We design the following experiments in order to benchmark mac3 and mac3cache. The main purpose is to test how the cache performs in the case where a constraint check is extremely cheap (whose cost is the cost of function call + the cost of memory lookup, as mentioned in Section 2.4).

In implementing the cache, we try to make its access as fast as possible. When a constraint C_{XY} is revised, we first locate the cache area for C_{XY}, and before looking for a support for $a \in D_X$, we locate the cache area that stores the

relationship between a and values in D_Y. In this way, when checking a against a value $b \in D_Y$, the value of b is used directly as an index to the cached content.

	mac3	mac3cache	MAC3	MAC3cache
P1 (easy problems)	0.56s	0.38s	0.62s	0.48s
P2 (hard problems)	382s	293s	474s	396s

Table 1. mac3 vs mac3cache. P1=$(50, 30, 150, 560)$. P2=$(50, 30, 150, 580)$. The number of constraint checks for P1 and P2 are 11.0M and 5.35B respectively.

From the experiment, we see that, although the constraint check is very cheap already, by using cache we are able to speed up mac3 by 30% for hard problems. This result implies that mac3 is very sensitive to the cost of constraint checks.

The idea of caching results of constraint checks is applicable to *mac* derived from most AC algorithms, with the exception of AC4, which builds complex data structures during its initialization phase and does not need to do more constraint checks afterward.

4 Exploiting the Residues

AC3 is one of the algorithms that simply revise a constraint without using any historical memory. Getting mac3 from from AC3 is straightforward. For algorithms that use auxiliary data structures however, there are two conventional methods. In this section we focus only on AC3.1.

The first approach involves re-initializing the structure and establishing supports from scratch every time the algorithm is invoked, in the same fashion as mac3. Even though the value remained in the auxiliary structure is not exploited to its full potential, this approache is widely used due to its simplicity and low overhead.

The second approach takes advantage of the value in the last structure carried over from the previous execution. Since the search for support proceeds only in one direction, in order for the algorithm to be correct we need to record all the past supports during search, so that we can start from the exact same state when backtrack. We will call this algorithm mac3.1. It is observed that the worst-case space complexity is not the same as AC3.1, which is $O(ed^2)$, but a larger $O(ed \min(n, d)$ [8]. A trace of the algorithm is given in the following example; it shows that mac3.1 cannot avoid both positive and negative repeat.

Example 1. Assume there are two neighboring variables X and Y, where $a \in D_X$, $D_Y = \{x, y, z, u, v\}$ ordered from left to right, and $C_{XY} = \{(a, y), (a, u)\}$. During an execution of mac3.1 (a, y) is checked and last(X,a,Y) becomes $[y]$ (the structure last is a now stack in which the top is the left-most value). Later, suppose that y is removed from D_Y; thus a new support for a must be found.

Consequently, (a, z) and (a, u) are checked and last(X,a,Y) becomes $[u, y]$. Now suppose that backtracking occurs and y is restored. As a result last(X,a,Y) is rolled back to its previous state of $[y]$. Suppose the process repeats: y is again removed and a new support for a must be found. (a, z) and (a, u) would be checked more than once.

4.1 Flexible Domain Search

The major problem with mac3.1 lie in its overhead in maintaining the auxiliary structure. This is necessary because the search always proceeds from the last support found. We can avoid this cost simply by restarting the search from the beginning again; however, this comes at the expense of optimality, since a constraint may be revised many times. We call this algorithm mac3.1residue. Note that in the pseudo-code we change the term from last to support to indicate that no invariant on its position is assumed.

Figure 2(i) shows the search for support from Example 1: suppose that the last support of a is y and that it is no longer available. The search would restart from the beginning until the next support (u) is reached. If u is later deleted then the entire domain must be searched. Note that this is conceptually the same as having a *circular* domain, although in practice it is easier to restart the search and have a normal domain.

mac3.1residue takes $O(ed^3)$ constraint checks in the worst-case while occupying $O(ed)$ space. It can avoid positive repeat but not negative repeat.

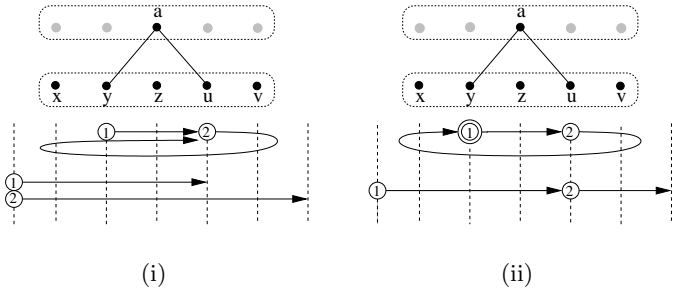


Fig. 2. Searching for support of a.

4.2 Optimal Worst-Case Flexible Domain Search

To make mac3.1residue optimal in the worst-case, we mark the first value that is checked when the constraint is revised for the first time in order to tell the search to stop as soon as this point is reached. This value, called start in the

mac3.1residue

```
preprocessing()
  1 AC3.1()
hasSupport(X, a, Y)
  2 b ← support(X, a, Y)
  3 if b ∈ D_Y then return true
  4 b ← head
  5 while b ← succ(b, D_Y) and b ≠ tail do
  6 |   if C_XY(a, b) then
  7 |   |   support(X, a, Y) ← b
  8 |   |   return true
  9 return false
```

pseudo-code, is initialized each time *mac* is called and it does not need to be reset when backtrack.

Figure 2(ii) shows how the search for support progresses. As in the case of mac3.1residue, we can implement this as a circular domain or starting out from the beginning, as shown in the bottom of Figure 2(ii). We choose the circular domain implementation because the supports found are more robust. Indeed, during search, *the deeper the level in which a support is found, the more robust it is* (also observed in [4].) This is due to fact a support found at a deeper level usually stays on in the current domain even after backtrack.

We call this algorithm mac3.1resOpt. It uses $O(ed)$ space in the worst-case. Like mac3.1residue, it can avoid positive repeat but not negative repeat.

mac3.1resOpt

```
preprocessing()
  1 AC3.1()
macInitialization()
  2 foreach C_XY ∈ C and a ∈ D_X and b ∈ D_Y do start(X, a, Y) ← support(X, a, Y)
hasSupport(X, a, Y)
  3 b ← support(X, a, Y)
  4 if b ∈ D_Y then return true
  5 while b ← cirSucc(b, D⁰_Y) and b ∈ D_Y and b ≠ start(X, a, Y) do
  6 |   if C_XY(a, b) then
  7 |   |   support(X, a, Y) ← b
  8 |   |   return true
  9 return false
```

The pseudo-code for mac3.1resOpt is presented in such a way that the idea is made as clear as possible; this should not be taken as the real implementation, especially the routine `macInitialization()`, in which each component of the structure start is reset. In fact we initialize start in a lazy way. Moreover, we only iterate through values in the current domain, rather than the original domain shown in line 6 of `hasSupport`. Using current domain involves more complex

terminating condition since value of $\mathsf{start}(X, a, Y)$ may not be in the current domain, thus rendering the condition $b \neq \mathsf{start}(X, a, Y)$ incorrect.

4.3 Experimental Results

The empirical performance of mac3.1, mac3.1residue, and mac3.1resOpt are shown in the following table. The running time for MAC includes that of its corresponding *mac*, e.g. for P1 the running time for MAC3.1 = 0.75s, 0.66s of which is the time taken up by mac3.1. There are no extra checks in MAC beyond *mac*.

	P1 (easy problems)		P2(hard problems)		P3(over-constrained)	
	#checks	time	#checks	time	#checks	time
mac3	11.0M	0.56s	5.35B	398s	10.36B	713s
mac3.1	7.3M	0.66s	3.46B	519s	6.38B	1185s
mac3.1resOpt	2.9M	0.55s	1.43B	472s	2.69B	668s
mac3.1residue	7.1M	0.46s	3.30B	310s	6.14B	465s
MAC3	-	0.62s	-	493s	-	848s
MAC3.1	-	0.75s	-	615s	-	1314s
MAC3.1resOpt	-	0.65s	-	561s	-	796s
MAC3.1residue	-	0.56s	-	400s	-	590s

Table 2. Performance of various algorithms. #checks = #constraint checks + #domain checks. P1 = $(50, 30, 150, 560)$. P2 = $(50, 30, 150, 580)$. The time for mac3.1 includes that for restoring the auxiliary data structure `last`.

From the table, it is not surprising to see both mac3.1residue and mac3.1resOpt do better than mac3.1 in all three categories, due to the absence of overhead in maintaining the auxiliary data structure. However, we observe two unexpected results. The first is that the non-optimal algorithm mac3.1residue conducts even fewer checks than the optimal mac3.1; it is worth emphasizing that optimality is a property of a single execution of arc consistency algorithm, whereas the data gathered in this experiments is accumulated over the entire course of solving a problem. The other is that mac3.1resOpt uses less than half the number of checks performed by mac3.1. This can be explained by the robustness of the residue.

mac3.1 is the slowest algorithm in the table. There are a few possible reasons. One is that the most frequent operations like value accessing and constraint checks are reasonably cheap (and thus the saving on checks does not compensate the cost). Another is that when we maintain its auxiliary data structures, we use a library class `stack`. The implementation could be made more efficient. There may exist more efficient ways to maintain `support` incrementally. However, the running time is very unlikely to be lower than those of *mac*using residues because of the larger number of checks.

mac3.1resOpt runs much slower than mac3.1residue although it conducts significantly less number of checks. The code at the innermost loop of our implementation of mac3.1resOpt is several times longer than that of mac3.1residue as a result of complex terminating condition remarked previously. The experiments show that the saving of checks does not compensate for the cost of the extra instructions in mac3.1resOpt. On the other hand, mac3.1residue is faster than mac3 because it takes advantage of the last residue without incurring the heavy overhead associated with mac3.1. In fact, the code of routine `hasSupport` for mac3.1residue is only a few more instructions than that of mac3.

5 Adaptive Domain Ordering

Some AC algorithms like AC3.1 demand an ordering on the values of the domains of variables. The key idea behind is to find a support of a value a of D_X with respect to a constraint C_{XY} by going through the values of the domain D_Y only once. For this purpose, an ordering of the values is needed so that we never make the same constraint check twice when looking for a support for a during the execution of AC3.1. One implementation of AC3.1 in *mac* could assume a total ordering on the values of the domain of variables during the *whole* search procedure. Since the values in a domain are removed in an arbitrary order, it may be time-consuming to keep to the order when these values are restored.

As mentioned before, we implement a domain as a doubly linked list. When we put back a removed value back to a domain, to keep the correct ordering of the values, we need to go through the list and insert it in the correct position.

In this section, we use mac3.1resOpt as the basic algorithm. Our observation is that to keep the optimality of mac3.1resOpt, it is only necessary to guarantee the ordering of values in a domain in a single execution of *mac*. It implies that when we restore a domain at backtracking or failure of assignment, we can simply append the pruned values to the linked list. The additional benefit is that all negative repeat can be avoided.

We test the idea on random problems. In the following table, mac-order is mac3.1resOpt where the ordering of values are kept during the whole search process by using `restoreDomain-order`, and mac-tail is mac3.1resOpt in which pruned values are appended to the end of the domains by `restoreDomain-tail`.

Even though appending the removed values to the end of a domain is 20% better in term of runing time than inserting the removed values to a domain according to the ordering, the performance of mac-tail is, surprisingly, inferior to that of mac-order. One explanation is that mac-order is optimized over the entire period of search, which can not be done for mac-tail since the ordering on domain values is different from one execution to another.

6 Related Works

In boolean satisfiability problem, the time needed to find the clause suitable for unit propagation is recognized to be the most expensive part. In [6] the authors

	P1 (easy problems)		P2(hard problems)	
	#checks	time	#checks	time
restoreDomain-order	-	0.10s	-	92s
restoreDomain-tail	-	0.06s	-	77s
mac-order	2.88M	0.55s	1.43B	494s
mac-tail	2.87M	0.70s	1.42B	525s
MAC-order	-	0.65s	-	588s
MAC-tail	-	0.77s	-	605s

Table 3. Performance of adaptive domain algorithm.

suggest that the solver keep track of two literals (called *watched literals*) so that a single unvalued literal could be detected quickly.

Watched literals and the variants of *mac* presented here have some important features in common. First, the search direction is not fixed. Second, nothing needs to be restored upon backtracking.

7 Conclusions

In this paper we propose studying *mac* and AC algorithms separately. Given the observation that *mac* will be executed many times, we have presented a few strategies that improve the efficiency of *mac* algorithms. They are analyzed in terms of time complexity, space complexity, number of checks (including constraint checks and domain checks), and running time.

First we study *mac* that caches the results of constraint checks. This proposal applies to most *mac* except those derived from AC4. On the one hand, the cache makes the cost of a constraint check almost constant over long period. Space permitting, most *mac* algorithms could be equipped with a cache. This would make the comparison of the performance of different *mac* algorithms more meaningful. The space complexity of the cache is the same as that of *mac* derived from an optimal AC algorithm. One the other hand, the cache also speeds up the running time of *mac*. In our experiment on hard problems, mac3cache saves 20% to 30% running time even under cheap constraint checks. On hard problems, mac3cache is the fastest among all algorithms reported in this paper. Given sufficient RAM, mac3cache is a good choice given its efficiency and simplicity.

For *mac* based on AC algorithms that use auxiliary data structures, we propose two schemes that reuse the residual value from previous execution: one keeps the worst-case running time optimal while the other concerns only about simplicity. Specifically, we present algorithm mac3.1resOpt using the former scheme and mac3.1residue using the latter. The conventional *mac* based on AC3.1 takes $O(ed\min(n,d))$ worst-case space complexity while mac3.1resOpt and mac3.1residue takes only $O(ed)$. Moreover, both algorithms do not maintain the last structure, thus avoiding the costly overhead suffered by mac3.1. An unexpected discovery through our experiment is that the robustness of residue

play a major role in the efficiency of *mac*. The number of checks performed by mac3.1residue and mac3.1resOpt are significantly lower than those of mac3 and mac3.1 respectively (mac3.1residue performs even fewer checks than mac3.1). It is worth emphasizing that although mac3.1residue has a worst-case complexity of $O(ed^3)$, it is the fastest among *mac* algorithms that have $O(ed)$ space complexity. Moreover, mac3.1residue is just as easy to implement as mac3. In retrospect, mac3.1residue can also be understood as a *mac* algorithm that uses a much smaller cache, recording only a single support for each value with respect to affiliated constraints.

We could apply the same idea to non-binary constraint networks. AC3.2 [4] generalizes AC3.1 to address non-binary constraints and takes advantage of positive multi-directionality by setting the current support found, which is a tuple for non-binary CSPs, as an *external support* for all other values in that tuple. This requires an extra storage apart from last because if it were to be used for this purpose then some values could be overlooked due to the fixed direction and range for support search. By using circular domain last can be used to store external support as well. It remains to be seen how this approach compares to AC3.2 and AC3.3, which counts all external supports and requires maintenance.

Finally we introduce adaptive domain ordering for *mac*, which in theory can avoid all negative repeat. In practice however, our experiment shows that *mac* that uses adaptive domain is worse off than that one that uses ordinary domain.

The lesson learned here is that consistency processing in the context of search deserved to be studied separately from its standalone version.

References

1. Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, Washington, DC, USA, 1993.
2. Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI-01*, pages 309–315, Seattle, Washington, 2001.
3. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.
4. Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP-03*, pages 480–494, Kinsale, Ireland, 2003.
5. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
6. M. Moskewisz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *The 39th Design Automation Conference*, 2001.
7. Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, Amsterdam, The Netherlands, 1994.
8. M. R. C. van Dongen. To avoid repeating checks does not always save time. In *Proceedings of AICS'2003: The 14th Irish Artificial Intelligence and Cognitive Science*, Dublin, Ireland, 2003.

9. D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report MAC-AI-TR-271, MIT, Cambridge, MA, 1972.
10. Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of IJCAI-01*, pages 316–321, Seattle, Washington, 2001.

Two New Lightweight Arc Consistency Algorithms

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

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

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

1 Introduction

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

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

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

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

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

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

2 Constraint Satisfaction

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

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

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

3 Related Literature

3.1 Introduction

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

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

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

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

3.2 Revision Ordering Heuristics

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

Arc based revision ordering heuristics are the most commonly presented. Given some selection criterion they select an arc (v, w) for the next revision. For this class of heuristics candidate arcs are stored in a data structure called a *queue*, which usually corresponds to a set or a list. Selecting the best arc from the queue can be expensive because of the following reasons. Selecting the best arc from a set or a list based queue requires $\mathcal{O}(e)$ time. However, [16] describes selection criteria for which the queue can be represented by more efficient data structures facilitating a more efficient $\mathcal{O}(n)$ selection. Second, each selected arc corresponds to exactly one revision. Since there may be

many revisions there may be many selections. However in some cases like in AC-3_d or in AC-3_p when the reverse arc is present in the queue it allows for two simultaneous revisions. Arc based revision ordering heuristics update a queue after every effective revision, and many updates can be an overhead too.

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

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

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

3.3 AC-3 with Reverse Variable based Revision Ordering Heuristics

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

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

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

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

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

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

Fig. 2. Algorithm *revise*

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

4.1 Introduction

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

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

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

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

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

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

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

4.2 AC-3_{dl}

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

find a support in $D(w)$ then the value c of each previous neighbour w already known to support this deleted value r in $D(v)$ is marked $unsupported$.

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

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

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

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

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

4.3 AC-3_{ds}

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

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

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

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

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

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

5 Experimental Results

5.1 Introduction

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

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

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

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

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

5.2 Stand-alone Arc Consistency

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

Note that for a variable based heuristic, the number of selections of an element from the queue is very low. This is because when a variable is selected from the queue it allows to perform a number of revisions which is between 1 and the current degree

of the variable. On the other side for an arc based heuristic, the number of selections of an element from the queue is high. The reason is that each arc that is selected from the queue corresponds to exactly one revision. However for AC-3_d it is somewhat reduced because sometimes AC-3_d performs two revisions per selection.

Table 1. Average results for random problems P3

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

Table 2. Average results for random problems P4

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

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

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

5.3 Maintaining Arc Consistency during Search

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

Table 3. Average Results for random problems of size 25

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6 Conclusion

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

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

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

References

1. C. Bessière and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of the* 11^{th} *National Conference on Artificial Intelligence (AAAI'93)*, Washington, DC, 1993.
2. C. Bessière, E.C. Freuder, and J.-C. Régin. Using inference to reduce arc consistency computation. In C.S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 592–598, Montréal, Québec, Canada, 1995. Morgan Kaufmann Publishers.
3. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 309–315, 2001.
4. B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Journal of Constraints*, 4:79–89, 1999.
5. J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceeding of the* 2^{nd} *Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.
6. I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 327–340. Springer, 1997.
7. I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.
8. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
9. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–73, 1985.
10. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
11. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
12. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.
13. M.R.C. van Dongen. AC-3_d an efficient arc-consistency algorithm with a low space-complexity. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture notes in Computer Science*, pages 755–760. Springer, 2002.
14. M.R.C. van Dongen. Improving MAC by sacrificing the optimality of the worst case time-complexity of its arc-consistency component, 2004. In preparation.
15. M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 2004. Accepted for publication.
16. M.R.C. van Dongen and D. Mehta. Queue representation for arc consistency algorithms, 2004. Submitted for publication.
17. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *AI/GI/VI '92*, pages 163–169, Vancouver, British Columbia, Canada, 1992.
18. Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 316–321, 2001.

Maintaining Arc Consistency Algorithms During the Search with an Optimal Time and Space Complexity

Jean-Charles Régin

ILOG Sophia Antipolis, Les Taissounières HB2, 1681 route des Dolines, 06560 Valbonne, France, email: regin@ilog.fr

Abstract. In this paper, we present in detail the versions of the arc consistency algorithms for binary constraints based on list of supports and last value when they are maintained during the search for solutions. In other words, we give the explicit codes of MAC-6 and MAC-7 algorithms. Moreover, we present an original way to restore the last values of AC-6 and AC-7 algorithms in order to obtain a MAC version of these algorithms whose space complexity remains in $O(ed)$ while keeping the $O(ed^2)$ time complexity on any branch of the tree search. This result outperforms all previous studies.

1 Introduction

In this paper we focus our attention on binary constraints. For more than twenty years, a lot of algorithms establishing arc consistency (AC algorithms) have been proposed: AC-3 [6], AC-4 [7], AC-5 [12], AC-6 [1], AC-7, AC-Inference, AC-Identical [2], AC-8 [4], AC-2000: [3], AC-2001 (also denoted by AC-3.1 [13]) [3], AC-3_d [11], AC-3.2 and AC-3.3 [5].

The MAC version of an AC algorithm, is the maintain of the algorithm during the search for a solution.

Some MAC versions of AC algorithms are easy, like AC-3 or AC-2000. Some others AC algorithms are much more complex to be maintained during the search. This is mainly the case for algorithms based on the notion of list of support (S-list) and on the notion of last support (last value). These algorithms, like AC-6, AC-7, AC-2001, AC-Inference, involve some data structures that need to be restored after a backtrack. Currently, there is no MAC version of these algorithms capable to keep the optimal time complexity on every branch of the tree search ($O(d^2)$ per constraint, where d is the size of the largest domain), without sacrificing the space complexity. More precisely, the algorithms AC-6, AC-7 and AC-2001 involve data structures that lead to a space complexity of $O(d)$ per constraint, but the MAC versions of these algorithms require to save some modifications of these data structures in order to restart the computations after a backtrack in a way similar as if this backtrack did not happen, and so they keep the same time complexity for any branch of the tree search as for one establishment of arc consistency. These savings have a cost which depends

on the depth of the tree search and that are bounded by d. Therefore for these reasons some authors have proposed algorithms having a $O(d\min(n,d))$ space complexity per constraint [8–10], thus the nice space complexity of these AC algorithms is lost for their MAC versions.

In this paper, we propose an original MAC version of the algorithms involving S-list and last data with a space complexity in $O(d)$ per constraint while keeping the optimal time complexity ($O(d^2)$) for any branch of the tree search.

At this moment, our goal is not to propose an algorithm that outperforms MAC-6, MAC-7 or MAC-2001, but to solve an open question. The capability to avoid to need some extra data can also be quite important, for embedded systems for instance, where all the possible memory requirements must be precomputed and reserved.

This paper is organized as follows. First, we recall some definitions of CP and we give a classical backtrack algorithm associated with a propagation mechanism. Then, we give a classical AC algorithm using the S-List and last data structures. Next, we identify the problems of the MAC version of this algorithm, and we propose a new MAC version optimal in time and in space. At last, we conclude.

2 Preliminaries

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. We introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of definition domains of $\mathcal{N}$. Indeed, we consider that any constraint network $\mathcal{N}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $T(C)$ is called a **tuple on** $X(C)$ and r is the **arity** of C and $\tau[x]$ denotes the value of the variable x in the tuple τ.

A value a for a variable x is often denoted by (x, a). Then, (x, a) is **valid** if $a \in D(x)$, and a tuple is **valid** if all the values it contains are valid. Let C be a constraint. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $a = \tau[x]$ (τ is the called a **support** for (x, a) on C.) A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

A **filtering algorithm** associated with a constraint C is an algorithm which may remove some values that are inconsistent with C; and that does not remove any consistent values. If the filtering algorithm removes all the values inconsistent with C we say that it establishes the arc consistency of C.

Propagation is the mechanism that consists of calling the filtering algorithm associated with the constraints involving a variable x each time the domain of this variable is modified. Note that if the domains of the variables are finite, then this process terminates because a domain can be modified only a finite number of times. The set of values that have been removed from the domain of a variable x is called **the delta domain** of x. This set is denoted by $\Delta(x)$ [12].

Algorithm 1: function searchForSolution

```
PROPAGATION()
    while ∃y such that Δ(y) ≠ ∅ do
        pick y with Δ(y) ≠ ∅
        for each constraint C involving y do
            if ¬ FILTER(C, x, y, Δ(y)) then return false
        RESET(Δ(y))
    return true
SEARCHFORSOLUTION(x, a)
    ADDCONSTRAINT(x = a)
    if all variables are instantiated then PRINTSOLUTION()
    else
        if PROPAGATION() then
            do
                y ← SELECTVARIABLE()
                b ← SELECTVALUE(y)
                SEARCHFORSOLUTION(y, b)
                REMOVEFROMDOMAIN(y, b)
            while D(y) ≠ ∅ and PROPAGATION()
    RESTORECN()
```

Function PROPAGATION of Algorithm 1 is a possible implementation of this mechanism. The filtering algorithm associated with the constraint C defined on x and y corresponds to Function FILTER$(C, x, y, \Delta(y))$. This function removes the values of $D(x)$ that are not consistent with the constraint in regards to $\Delta(y)$. For a constraint C this function will also be called with the parameters $(C, y, x, \Delta(x))$. We also assume that function RESET$(\Delta(y))$ is available. This function sets $\Delta(y)$ to the empty set. The algorithm we propose is given as example, and some other could be designed. For our purpose, we only suppose that the delta domain is available.

Algorithm 1 also contains a classical search procedure (a backtrack algorithm) which selects a variable, then a value for this variable and call the propagation mechanism. Note that at the end of the recursive function SEARCHFORSOLUTION, Function RESTORECN is called. This function restores the data structures used by the constraint when a backtrack occurs. We assume that Function

searchForSolution is called first with a dummy variable x and a dummy value a such that the constraint $x = a$ has no effect.

3 Arc consistency algorithms

Consider a constraint C defined on x and y for which we study the consequences of the modifications of the domain of y.

Algorithm 2: An AC algorithm

```
filter(in C, x, y, Δ(y)): boolean
    (x, a) ← firstPendingValue(C, x, y, Δ(y))
    while (x, a) ≠ nil do
        if ¬ existValidSupport(C, x, a, y, getΔValue(C)) then
1           removeFromDomain(x, a)
            if D(x) = ∅ then return false
        (x, a) ← nextPendingValue(C, x, y, Δ(y), a)
    return true
```

Definition 1 *We call* **pending values** *w.r.t. the variable y the set of* **valid** *values of a variable x for which a support is sought by an AC algorithm when the consequences of the deletion of the values of a variable y are studied.*

Thanks to this definition, the principles of AC algorithms can be easily expressed: **Check whether there exists a support for every pending value and remove those that have none**.
Algorithm 2 is a possible implementation of this principle.

We can now give the principles of Functions firstPendingValue, nextPendingValue and of Function existValidSupport for AC-6 and AC-7 algorithm. Since we consider a constraint C involving x and y and that y is modified, then the pending values belong only to x.

AC-6: AC-6 was a major step in the understanding of the AC-algorithm principles. AC-6 mixes some principles of AC-3 with some ideas of AC-4. AC-6 can be viewed as a lazy computation of supports. AC-6 introduces the **S-list** data structure: for every value (y, b), the S-list associated with (y, b), denoted by S-list$[(y, b)]$, is the list of values that are currently supported by (y, b). In AC-6 the knowledge of only one support is enough, then a value (x, a) is supported by **only one** value of $D(y)$, so there is only value of $D(y)$ that contains (x, a) in its S-list. Then, the pending values are the valid values contained in the S-lists of the values in $\Delta(y)$. Function existValidSupport is an improvement of the AC-3's one, because the checks in the domains are made w.r.t an ordering and are started from the support that just has been lost, which is the delta value containing the current value in its S-list. The space complexity of AC-6 is in

$O(d)$ and its time complexity is in $O(d^2)$.
AC-7: This is an improvement of AC-6. AC-7 exploits the fact that if (x, a) is supported by (y, b) then (y, b) is also supported by (x, a). Thus, when searching for a support for (x, a), AC-7 proposes, first, to search for a valid value in S-list$[(x, a)]$, and every non valid value which is reached is removed from the S-list. We say that the support is sought by **inference**. This idea contradicts an invariant of AC-6: a support found by inference is no longer necessarily the latest checked value in $D(y)$. Therefore, AC-7 introduces explicitly the notion of **latest checked value** by the data **last** associated with every value. AC-7 ensures the property: If last$[(x, a)] = (y, b)$ then there is no support (y, a) in $D(y)$ with $a < b$. If no support is found by inference, then AC-7 uses an improvement of the AC-6's method to find a support in $D(y)$. When we want to know whether (y, b) is a support of (x, a), we can immediately give a negative answer if last$[(y, b)] > (x, a)$, because in this case we know that (x, a) is a not a support of (y, b) and so that (y, b) is not a support for (x, a). The properties on which AC-7 is based are often called bidirectionnalities. Hence, AC-7 is able to save some checks in the domain in regards to AC-6, while keeping the same space and time complexity.

The MAC version of AC-6 needs an explicit representation of the lastest checked value, thus the AC-6 and AC-7 algorithms use the following data structures:
Last: the last value of (x, a) for a constraint C is represented by last$[(x, a)]$ which is equals to a value of y or nil.
S-List: these are classical list data structures.

Algorithm 3: Pending values computation.

```
FIRSTPENDINGVALUE(C, x, y, Δ(y)): value
    b ←FIRST(Δ(y))
    return TRAVERSES-LIST(C, x, y, Δ(y))
NEXTPENDINGVALUE(C, x, y, Δ(y), a): value
    return TRAVERSES-LIST(C, x, y, Δ(y))
TRAVERSES-LIST(C, x, y, Δ(y)): value
    while (y, b) ≠ nil do
        (x, a) ←SEEKVALIDSUPPORTEDVALUE(C, y, b)
        if (x, a) ≠ nil then return (x, a)
        b ←NEXT(Δ(y), b)
    return nil
GETΔVALUE(C, x, y, Δ(y)): return b
```

We can give a MAC version of AC-6 and AC-7 :

Algorithm 3 is a possible implementation of the computation of pending values. Note that some functions require "internal data" (a data whose value

is stored). We assume that FIRST$(D(x))$ returns the first value of $D(x)$ and NEXT$(D(x), a)$ returns the first value of $D(x)$ strictly greater than a.

Function SEEKVALIDSUPPORTEDVALUE(C, x, a) returns a valid supported value belonging to the S-list$[(y, b)]$ (see Algorithm 6.)

Algorithm 4: Function EXISTVALIDSUPPORT

```
EXISTVALIDSUPPORT(C, x, a, y, δy): boolean
if last[(x, a)] ∈ D(y) then (y, b) ← last[(x, a)])
if AC-7 and (y, b) = nil then
    (y, b) ← SEEKVALIDSUPPORTEDVALUE(C, x, a)
if (y, b) = nil then (y, b) ← SEEKSUPPORT(C, x, a, y)
UPDATES-LIST(C, x, a, y, δy, b)
return ((y, b) ≠ nil)
```

Algorithm 5: Functions seeking for a valid support

```
SEEKSUPPORT(C, x, a, y) : value
    b ←NEXT(D(y),last[(x, a)])
    while b ≠ nil do
        if last[(y, b)] ≤ (x, a) and ((x, a), (y, b)) ∈ T(C) then
            last[(x, a)] ← (y, b)
            return (y, b)
        b ←NEXT(D(y), b)
    return nil
```

Algorithm 4 is a possible implementation of Function EXISTVALIDSUPPORT and Algorithm 5 is a possible implementation of Function SEEKSUPPORT.

The S-list representation will be detailed in a specific section, notably because it has been careful designed in order to be maintained during the search.

4 Maintain during the search

The management of the AC algorithms has been studied in detail in [8].

Two types of data structures can be identified for a filtering algorithm (like an AC algorithm for instance) :

• the **external data structures**. These are the data structures from which the constraint of the filtering algorithm is stated, for instance the variables on which the constraint is defined or the list of allowed combinations by the constraint.

• the **internal data structures**. These are the data structures needed by the filtering algorithm. The space complexity of the filtering algorithm is usually based on these data structures. For instance, AC-6 and AC-7 require data

structures in $O(d)$.

We will say that the space complexity of a MAC version of an AC algorithm is optimal if it is the same as the space complexity of the AC algorithm.

In this section, we propose a MAC version of an AC algorithm using S-list and/or last value having an optimal space and time complexity.

There is no particular problem when we go down to the search tree, because the instantiation of new variable leads only to the deletion of values. The main difficulty is to manage the data structures when a failure occurs, that is when there is a backtrack.

Consider that n is the current node of the search. The data structures associated with an AC algorithm contain certain values. These values are called the **state** of the data structures. Then, assume that the search is continued from n and then backtracked to n. In this case, two possibilities have been identified [8]:

- the state of the data structure at the node n is exactly restored
- an equivalent state is restored.

4.1 Exact restoration of the state

This method saves the modifications of the state of an AC algorithm in order to restore exactly this state after a backtrack. In other word, every data contains the same value as it had when n was the current node. This implies that every modification of the value of a data has to saved in order to be restored after a backtrack. Every S-list and every last value can be modified d times per constraint during the search. Thus, the space complexity is multiplied by a factor of d. So, this possibility cannot lead to a MAC version with an optimal space complexity.

4.2 Restoration of an equivalent state

This is another notion which is based on the properties that have to be satisfied by the data structures. The algorithms have an optimal time complexity when some properties are satisfied. What is important is not the way they are satisfied, but only the fact that they are satisfied. For some data structures it is not necessary to restore exactly the values it contains before. For instance, if (y, b) was the current support of (x, a) for the node n and if this support changes to become (y, c) then (y, c) can be the current support of (x, a) when all the nodes following n are backtracked. This means that there is no need to change the elements in the S-list, no deletion is needed. It is only required to add some values that have been removed.

This method is much more interesting than the restoration of the exact state. We choose to use it and propose to study how we have to design and how we can manage the S-Lists and the last values in order to restore only an equivalent state, while keeping the optimal time complexity.

5 S-list management

If an equivalent state is accepted after backtracking, then when a support is modified there is no need to save it, because it does not need to be restored. However, in order to keep an optimal time complexity for every branch of the tree search, the MAC version of AC-6 and AC-7 algorithms needs to remove from S-lists the reached values that are not valid. In fact, it is necessary to avoid to consider them several times.

This is Function seekValidSupportedValue that manages this deletion. So, this function deserves a particular attention.

This function is called during the computation of the pending values or when a support is sought by inference (AC-7). When it is called for a value (x, a) it traverses the S-list of (x, a) until a valid value is found and removes from this S-list the value that are reached and not valid. In the MAC version, a restoration of the S-list is obviously needed. More precisely, if (y, b) is reached when traversing S-list$[(x, a)]$ this means that (x, a) is the current support of (y, b) for this constraint. Thus, if (y, b) is no longer valid when it is reached, then (y, b) is removed from S-list$[(x, a)]$, but after backtracking the node of the tree search that led to the deletion of (y, b) it is necessary to restore (y, b) in S-list$[(x, a)]$, because at this moment (y, b) will be valid and (y, b) needs to have a support. Therefore, when an element is removed from the S-list when traversing it, it is necessary to save this information in order to restore it later.

In order to avoid unnecessary memory consumption we propose to represent the S-list as follows :

- The first element of an S-list of a value (y, b) is denoted by firstS$[C, (y, b)]$ which is equals to a value of x or nil.
- The S-lists exploit the fact that for a constraint, each value (x, a) can be in at most one S-list. So, every value (x, a) is associated with a data nextInS$[C, (x, a)]$ which is the next element in the S-list of the support of (x, a). For instance, S-list$[C, (y, b)] = ((x, a), (x, d), (x, e))$ will be represented by : firstS$[C, (y, b)] = (x, a)$; nextInS$[C, (x, a)] = (x, d)$; nextInS$[C, (x, d)] = (x, e)$; nextInS$[C, (x, e)] = nil$. The nextInS data are systematically associated with every value so they are **preallocated**.

The saving/restoration of a support by MAC, can be easily done by adding a data to every value (x, a): restoreSupport$[C, (x, a)]$. This data contains the support of (x, a) if (x, a) has been removed from the S-list of its support; otherwise it contains nil. This data will be used to restore (x, a) in the S-list of its support when (x, a) will be restored in its domain. More precisely, assume that (y, b) is the support of (x, a) and that (x, a) has been removed from S-list$[C, (y, b)]$ when searching for a valid support of (y, b) by inference. In this case, the data restoreSupport$[C, (x, a)]$ will be set to (y, b). And, when (x, a) will be restored in the domain of its variable after backtracking, then (x, a) will be added to the S-list of restoreSupport$[C, (x, a)]$. Of course, if restoreSupport$[C, (x, a)]$ is nil then nothing happens.

This mechanism of restoration implies that a solver has the capability to perform some operations when a value is restored in its domain. This is not a strong assumption and can be made with all existing solvers.

Another point must also be considered. Function existValidSupport calls Function updateS-list in order to update the S-list when a new valid support is found. Conceptually there is no problem, if a new valid support (y, b) is found for a value (x, a) then (x, a) is added to S-list$[(y, b)]$. However, before being added to the S-list (x, a) must be removed from the S-list of its current support. This deletion causes some problems of implementation because the S-lists are simple lists and to perform a deletion it is necessary to know the previous element. In order to avoid this problem, we have decided to systematically remove all the reached elements from the S-list. Thus, every element which is considered is the first element of the list and so there is no longer any problem to remove it. If a new support is found then the element can be safely added to a new S-list. If there is no valid support then it will be necessary to restore (x, a) in the S-list of its support. this result can be easily obtained by using the previous mechanism of saving/restoration. Function updateS-list implements that idea. Algorithm

Algorithm 6: Management of Supported Values Lists

```
seekValidSupportedValue(C, x, a) : value
    while firstS[C, (x, a)] ≠ nil do
        (y, b) ← firstS[C, (x, a)]
        if b ∈ D(y) then return (y, b)
        firstS[C, (x, a)] ← nextInS[C, (y, b)]
        restoreSupport[C, (y, b)] ← (x, a)
    return nil
updateS-list(C, x, a, y, δy, b)
    firstS[C, (y, δy)] ← nextInS[C, (x, a)]
    if (y, b) = nil then restoreSupport[C, (x, a)] ← (y, δy)
    else
        nextInS[C, (x, a)] ← firstS[C, (y, b)]
        firstS[C, (y, b)] ← (x, a)
restoreSupports(C, (x, a))
    // the value a is restored in D(x)
    (y, b) ← restoreSupport[C, (x, a)]
    if restoreSupport[C, (x, a)] ≠ nil then
        // (x, a) is added to the S-list of its support (y, b)
        nextInS[C, (x, a)] ← firstS[C, (y, b)]
        firstS[C, (y, b)] ← (x, a)
        restoreSupport[C, (x, a)] ← nil
```

6 gives a possible implementation of the management of S-lists.

6 Last management

The notion of latest checked value (last value) is necessary for AC-6 and AC-7 algorithms to have an $O(d^2)$ time complexity per constraint.

The last value satisfies two properties :

Property 2 *Let* $(y, b) = last[C, (x, a)]$ *then*
$\forall c \in D(y), c < b \Rightarrow ((x, a), (y, c)) \notin T(C)$.

This first property ensures that there is no reason to consider again the values that are less than the last value.

Property 3 *Let* $(y, b) = last[C, (x, a)]$ *then*
Function SEEKSUPPORT *has never checked the compatibility between* (x, a) *and any element* $d \in D(y)$ *with* $d > b$.

This property ensures that no compatibility check has been performed for the values of y greater than the last value.

These two properties ensures that the compatibility between two values will never be checked twice (if the bidirectionnality is not taken into account).

If the last are not restored after backtracking then the time complexity of AC-6 and AC-7 algorithms is in $O(d^3)$. We can prove that claim with the following example. Consider a value (x, a) that has exactly ten supports among the 100 values of y: $(y, 91), (y, 92)..., (y, 100)$; and a node n of the tree search for which the support of (x, a) is $(y, 91)$. If the last value is not restored after a backtrack then there are two possibilities to define a new last value:

1. the new last value is recomputed from the first value of the domain
2. the new last value is defined from the current last value, but the domains are considered as circular domains: the next value of the maximum value is the minimum value.

For the first case, it is clear that all the values strictly less than $(y, 91)$ will have to be reconsidered after every backtrack for computing a valid support.
For the second possibility, we can imagine an example for which before backtracking $(y, 100)$ is the current support; then after the backtrack and since the domains are considered as circular domains it will be necessary to reconsidered again all the values that are strictly less than $(y, 91)$.

Note also, that if the last value is not correctly restored it is no longer possible to totally exploit the bidirectionnality. So, it is necessary to correctly restored the last value.

6.1 Saving-Restoration of Last

The simplest way is to save the current value of last each time it is modified and then to restore these values after a backtrack. This method can be improved by remarking that it is sufficient to save only the first modification of the last value for a given node of the tree search. In this case, the space complexity of AC-6 and AC-7 algorithms is multiplied by $\min(n, d)$ where n is the maximum of the tree search depth [8–10].

6.2 Recomputation of last

First, with the restoration of an equivalent state, it is necessary to slightly modify the algorithm. The last value of (x, a) can indeed be valid and not be the current support of (x, a), because the current support has been found by inference and no support is restored. Thus, it is necessary to check the validity of the last value in the MAC version of an AC algorithm.

Now, we propose an original method to restore the correct last value. Instead of being based on savings this method is based on recomputation.

The idea of this algorithm is quite simple. Assume that we backtrack from a node n, and consider a variable x. We will denote by $D_R(y)$ the values of the variable y that have to be restored during this backtrack. Then, we have the following proposition on which our algorithm is based:

Proposition 4 *Let (x, a) be a value, and $T(C, (x, a), D_R(y))$ be the set of values of $D_R(y)$ that are compatible with (x, a) w.r.t C. Then* $\min(last[C, (x, a)], \min(T(C, (x, a), D_R(y))))$ *is a possible value of $last[C, (x, a)]$.*

proof: To prove this proposition we need to prove that $\min(\text{last}[C, (x, a)], \min(T(C, (x, a), D_R(y))))$ satisfies both Property 2 and Property 3:

• Property 2:

Let $D_A(y)$ be the domain of the variable y after the backtrack, and $D_B(y)$ be the domain of the variable y before the backtrack. Since the algorithm systematically checked if the last value is valid, then after the backtrack the last value of (x, a) will be the minimum between its current value and the value $b \in D_A(y)$ such that (x, a) and (y, b) are compatible and there is no value $c \in D_A(y)$ with $c < b$ that is compatible with (x, a). We have $D_A(y) = D_B(y) \cup D_R(y)$. Then before the restoration either $\text{last}[C, (x, a)] \in D_B(y)$ or $\text{last}[C, (x, a)] \in D_R(y)$ or $\text{last}[C, (x, a)] \notin D_A(y)$. Moreover, if $\text{last}[C, (x, a)] \in D_B(y)$ before the restoration, then it does not exist another value $c \in D_B(y)$ compatible with (x, a) and such that $(y, c) < \text{last}[C, (x, a)]$ by definition of last. Hence, to compute the minimum, it is enough to compare $\text{last}[C, (x, a)]$ with only the compatible values of $D_R(y)$, and Property 2 is satisfied.

• Property 3:

The last value of (x, a) can only be equal to either nil or a value of y which is compatible with (x, a), by definition. Then, $\min(\text{last}[C, (x, a)], \min(T(C, (x, a), D_R(y))))$ is either equal to $\text{last}[C, (x, a)]$ or equal to $\min(T(C, (x, a), D_R(y)))$. The first case means that the last value has not been modified after node n then Property 2 is satisfied from the branch of the tree search going from the root to the node n after the backtrack to node n. The second case means that the smallest compatible value b of $D(y)$ after the backtrack to node n is the new last value. Suppose that Function seekSupport has reached a value c of y when seeking for a new support for the value (x, a) in the branch of the tree search going from the root to node n. At the node n the value a belongs to the domain of y, so it means that a support d greater than c has been found by Function seekSupport and that a last value greater than c exists. Since $b = \min(T(C, (x, a), D_R(y)))$ is

the smallest valid value of y compatible with (x, a), the value d with $d > b$ cannot be a last value, so it is impossible to find such a value and Property 3 holds. $\odot$

Only the values of $D(x) \cup D_R(x)$ needs to have their last value restored. So, we obtain the new algorithm (see Algorithm 7.) Function RECOMPUTELAST is called for every variable of every constraint after every backtrack.

Algorithm 7: Restoration of last by recomputation

```
RECOMPUTELAST(C, x)
for each a ∈ (D(x) ∪ D_R(x)) do
    for each b ∈ D_R(y) do
        if ((x, a), (y, b)) ∈ T(C) then
            last[C, (x, a)] ← min(last[C, (x, a)], (y, b))
```

It is important to note that it is possible to recompute a value of last which is greater than the last value that would had been restored by using the saving/restoration mecanism. So, during the backtrack to the node n we can benefit from the computations that have been made after the node n.

Let us study the time complexity for any branch of the tree search, that is when we backtrack from a leaf to the root.
For one restoration and for one variable of a constraint the time complexity of Function RECOMPUTELAST is in $O(|D(x)| \times |D_R(y)|)$.
For one branch of the tree search the time complexity of the calls of Function 7 is in $O(\sum_{D_{Ri}} |D_0(x)| \times |D_{Ri}(y)|) = O(|D_0(x)| \times \sum_{D_{Ri}} |D_{Ri}(y)|)$. Moreover, the set $D_{Ri}(y)$ are pairwise disjoint for one branch of the tree search and their union is included in $D(y)$. Therefore we have $\sum_{D_{Ri}} |D_{Ri}(y)| \leq |D_0(y)|$ and the time complexity is in $O(|D_0(x)| \times |D_0(y)|) = O(d^2)$ per constraint. So, we have the same time complexity as for AC-6 or AC-7 algorithms.

It is possible to give some improvements of the previous algorithm :

First, if the values of $D_R(y)$ are ordered then the number of tests can be limited, because the first compatible value which is less that the last value will become the new last value. If the complexity of one sort is in $d \log(d)$ then the time complexity of all the sorts for one branch of the tree search will be depends on $\sum_{i=k..1} |D_{Ri}(y)| \log(|D_{Ri}(y)|) \leq \sum_{i=k..1} |D_{Ri}(y)| \log(d) \leq \log(d) \sum_{i=k..1} |D_{Ri}(y)| \leq d \log(d)$. This number is less than d^2.

Then, we can separate the study of the values of x. There are several possibilities (note that $D(x)$ and $D_R(x)$ are disjoint) :

- $(x, a) \in D_R(x)$. It is possible to use a new data that stores the first value of a last for the current node (that is only one data is introduced per value). This new data saves the last value that have to be restored for the values that are removed by the current node. So, the last of these values can be restored in $O(1)$ per value.

– $(x, a) \in D(x)$. In this case there are two possible cases :
  - $\text{last}[C, (x, a)] \notin D(y)$ and $\text{last}[C, (x, a)] \notin D_R(y)$. In this case the last is correct and no restoration is needed.
  - $\text{last}[C, (x, a)] \in D(y)$ or $\text{last}[(x, a)] \in D_R(y)$. There is no specific improvement : the systematic checks seem to be needed.

7 Conclusion

In this paper we have presented MAC versions of AC-6 and AC-7 algorithms. We have also given a way to restore the latest checked value that lead to MAC-6 and MAC-7 algorithms having the same space complexity as AC-6 and AC-7. This result improves all the previous studies.

References

1. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
2. C. Bessière, E.C. Freuder, and J-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.
3. C. Bessière and J-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA, 2001.
4. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):79–89, 1998.
5. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionnality in coarse-grained arc consistency algorithm. In *Proceedings CP'03*, pages 480–494, Cork, Ireland, 2003.
6. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
7. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
8. J-C. Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique.* PhD thesis, Université de Montpellier II, 1995.
9. M. van Dongen. Lightweight arc consistency algorithms. Technical Report TR-01-2003, Cork Constraint Computation Center, CS Department, University College Cork, Western Road, Cork, 2003.
10. M. van Dongen. Lightweight mac algorithms. Technical Report TR-02-2003, Cork Constraint Computation Center, CS Department, University College Cork, Western Road, Cork, 2003.
11. M.R. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings CP'02*, pages 755–760, Ithaca, NY, USA, 2002.
12. P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
13. Y. Zhang and R. Yap. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle, WA, USA, 2001.

CAC: A Configurable, Generic and Adaptive Arc Consistency Algorithm

Jean-Charles Régin

ILOG Sophia Antipolis, Les Taissounières HB2, 1681 route des Dolines, 06560 Valbonne, France, email: regin@ilog.fr

Abstract. In this paper, we present CAC, a new configurable, generic and adaptive algorithm for establishing arc consistency for binary constraints. CAC is configurable, that is by combining some parameters CAC corresponds to any existing AC algorithm: AC-3, AC-4, AC-6, AC-7, AC-2000, AC-2001, AC-8, AC-3_d, AC-3.2 and AC-3.3. CAC is generic, like AC-5, because it may takes into account the structure of the constraints. CAC is adaptive because the underlined algorithm can be changed during the computation in order to use the most efficient one. This new algorithm leads to a new nomenclature of the AC algorithms which is based on the different features used by the algorithm like the values that are reconsidered when a domain is modified, or the fact that bidirectionnality is taken into account, or the way a new support is sought. This new nomenclature shows that several new possible combinations are now possible. In other words, we can easily combine some ideas of AC-3 with some ideas of AC-7 and some ideas of AC-2001 with some ideas of AC-6. Some experimental results highlight the advantages of our approach.

1 Introduction

In this paper we focus our attention on binary constraints. For more than twenty years, a lot of algorithms establishing arc consistency (AC algorithms) have been proposed: AC-3 [6], AC-4 [7], AC-5 [9], AC-6 [1], AC-7, AC-Inference, AC-Identical [2], AC-8 [4], AC-2000: [3], AC-2001 (also denoted by AC-3.1 [10]) [3], AC-3_d [8], AC-3.2 and AC-3.3 [5]. Unfortunately, these algorithms are differently described and their comparison is not easy. Thus, we propose a configurable, generic and adaptive AC algorithm, called CAC.

Configurable means that the previous existing algorithms can be represented by setting some predefined parameters. This has some advantages:

• this unique algorithm can represent all known algorithms.

• it clearly shows the differences between all the algorithms. This extends the discussion started in [3].

• some new arc consistency algorithms can be easily and quickly derived from CAC, because some combinations of parameters have never been tested.

• CAC leads to a new nomenclature which is much more explicit than the current one ("AC-" followed by a number.), because algorithms are now ex-

pressed by combinations of predefined parameters. For instance, AC-3 is renamed: CAC-pvD-sD and AC-6 becomes CAC-pvΔs-last-sD.

Generic means that CAC is also a framework that can be derived to take into account some specificity of some binary constraints. In other word, dedicated algorithms can be written, for functional constraints for instance. This corresponds to a part of the generic aspects of AC-5. In our case, the incremental behavior of the AC-5 is generalized.

Adaptive means that CAC is able to use different algorithms successively as suggested in [3]. For instance, AC-2001 can be used then AC-7 and then AC-2001 depending on which one seems to be the best for the current configuration of domains and delta domains. We think, indeed, that **CP will be strongly improved if a filtering algorithm is in itself capable to select at each time its best version, instead of asking the user to do it a priori**.

AC-algorithms work in two steps. First, an initialization step is called. This step consists of finding a support (i.e. a compatible value) for each value. If a value has no support then it is removed from its domain. Then, in the second step the consequences of the deletion of a value are studied, that is a new support is sought for some values. In this paper, we will consider only the second step which is the most important.

This paper is organized as follows. First, we recall some definitions of CP. Then, we study all the existing algorithms, and we identify different concepts of the AC algorithms and detail CAC algorithm. A new nomenclature is proposed. Then, the adaptive behavior of CAC algorithm is considered. At last, after studying some experiments, we conclude.

2 Preliminaries

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $T(C)$ is called a **tuple on** $X(C)$ and $\tau[x]$ denotes the value of the variable x in the tuple τ. A value a for a variable x is often denoted by (x, a). (x, a) is **valid** if $a \in D(x)$, and a tuple is **valid** if all the values it contains are valid. Let C be a constraint. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $a\ \tau[x]$. A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

A **filtering algorithm** associated with a constraint C is an algorithm which may remove some values that are inconsistent with C; and that does not remove any consistent values. If the filtering algorithm removes all the values inconsistent with C we say that it establishes the arc consistency of C. **Propagation** is the mechanism that consists of calling the filtering algorithm associated with the constraints involving a variable x each time the domain of this variable

is modified. The set of values that have been removed from the domain of a variable x is called **the delta domain** of x. This set is denoted by $\Delta(x)$. More information about delta domains can be found in [9]. Function PROPAGATION of Algorithm 1 is a possible implementation of this mechanism. The filtering algorithm associated with the constraint C defined on x and y corresponds to Function FILTER$(C, x, y, \Delta(y))$. This function removes the values of $D(x)$ that are not consistent with the constraint in regards to $\Delta(y)$. For a constraint C this function will also be called with the parameters $(C, y, x, \Delta(x))$. We also assume that function RESET$(\Delta(y))$ is available. This function sets $\Delta(y)$ to the empty set. The algorithm we propose is given as example, and some others could be designed.

Algorithm 1: function PROPAGATION

```
PROPAGATION()
    while ∃y such that Δ(y) ≠ ∅ do
        pick y with Δ(y) ≠ ∅
        for each constraint C involving y do
            if ¬ FILTER(C, x, y, Δ(y)) then return false
        RESET(Δ(y))
    return true
```

3 Arc consistency algorithms

Consider a constraint C defined on x and y for which we study the consequences of the modification of the domain of y.

Algorithm 2: CAC filtering algorithm

```
FILTER(in C, x, y, Δ(y)): boolean
    get the parameters of C
    pvMode ← SELECTPENDINGVALUEMODE(C, x, y, Δ(y), pvMode)
    sMode ← SELECTEXISTSUPPORTMODE(C, x, y, Δ(y), sMode)
    (x, a) ← pvMode.FIRSTPENDINGVALUE(C, x, y, Δ(y))
    while (x, a) ≠ nil do
        if ¬ EXISTVALIDSUPPORT(C, x, a, y, sMode) then
1           REMOVEFROMDOMAIN(x, a)
            if D(x) = ∅ then return false
        (x, a) ← pvMode.NEXTPENDINGVALUE(C, x, y, Δ(y), a)
    return true
```

Definition 1 *We call* **pending values** *w.r.t. a variable y, the set of* **valid** *values of a variable x for which a support is sought by an AC algorithm when the consequences of the deletion of the values of the variable y are studied.*

Thanks to this definition, the principles of AC algorithms can be easily expressed: **Check whether there exists a support for every pending value and remove those that have none**.
Algorithm 2 is a possible implementation of this principle. This is also the core of the generic CAC algorithm. Functions selectPendingValueMode and selectExistSupportMode can be ignored at this point. They will be detailed later.

We can now give the principles of Functions firstPendingValue, nextPendingValue and of Function existValidSupport for each existing algorithm. We will consider a constraint C involving x and y and that y is modified. Therefore, the pending values belong only to x.
AC-3: The pending values are the values of $D(x)$, and $\Delta(y)$ is not used at all. All the values of $D(x)$ are considered and the search for a valid support is done by checking in $D(y)$ if there is a support for a value of $D(x)$. There is no memorization of the previous computations, so the same computations can be done several times and the time complexity for one constraint is in $O(d^3)$[1]. The advantage of this approach is that the space complexity is null.
AC-4: In AC-4 the tuple set is pre-computed and store in a structure that we call a **table**. This table contains for every value (x, a) a pointer to the next tuple involving (x, a). Therefore, the space complexity of AC-4 is in $O(d^2)$. The pending values are for each $(y, b) \in \Delta(y)$ all the valid values (x, a) such that $((x, a)(y, b)) \in T(C)$. Note that a value (x, a) can be considered several times as a pending value. The search for a support is immediate because Function existValidSupport can be implemented in $O(1)$ by associated with every value (x, a) a counter which counts the number of time (x, a) has a support in $D(y)$. Then, each time this function is called the counter is decremented (because (x, a) lost a valid support) and if the counter is equals to zero then there is no support. AC-4 was the first algorithm reaching an $O(d^2)$ time complexity, because no computation is made twice. However, a lot of computations are systematically done.
AC-5: This algorithm is mainly a generic algorithm. It has been designed in order to be able to take into account the specificity or the structure of the considered constraints. In other words, Function existValidSupport can be specialized by the user in order to benefit from the exploitation of the structure of the constraint. For instance, functional constraints are more much simple and arc consistency for these constraints can be achieved in $O(d)$ per constraint. Function filter and the propagation mechanism we gave are close to AC-5 ideas.
AC-6: AC-6 was a major step in the understanding of the AC-algorithm principles. AC-6 mixes some principles of AC-3 with some ideas of AC-4. AC-6 uses the idea of AC-4 to determine the pending values, but instead of considering all

[1] In this paper, we will always express the complexities per constraint, because a constraint network can involved several types of constraints. The usual way to express complexities can be obtained by multiplying the complexity we give by the number of binary constraints of the network.

the values supported by the values in $\Delta(y)$, it exploits the fact that the knowledge of one support is enough. AC-6 can be viewed as a lazy computation of supports. AC-6 introduces another data structure which is a variation of the table: the **S-list**: for every value (y, b), the S-list associated with (y, b), denoted by S-list$[(y, b)]$, is the list of values that are currently supported by (y, b). Contrary to AC-4, in AC-6 the knowledge of only one support is enough, then a value (x, a) is supported by **only one** value of $D(y)$, so there is only value of $D(y)$ that contains (x, a) in its S-list. Then, the pending values are the valid values contained in the S-lists of the values in $\Delta(y)$, and, for a given $\Delta(y)$, a value (x, a) can be considered only once as a pending value. Function EXISTVALIDSUPPORT is an improvement of the AC-3's one, because the checks in the domains are made w.r.t an ordering and are started from the support that just has been lost, which is the delta value containing the current value in its S-list. The space complexity of AC-6 is in $O(d)$ and its time complexity is in $O(d^2)$.

AC-7: This is an improvement of AC-6. AC-7 exploits the fact that if (x, a) is supported by (y, b) then (y, b) is also supported by (x, a). Thus, when searching for a support for (x, a), AC-7 proposes, first, to search for a valid value in S-list$[(x, a)]$, and every non valid value which is reached is removed from the S-list. We say that the support is sought by **inference**. This idea contradicts an invariant of AC-6: a support found by inference is no longer necessarily the latest checked value in $D(y)$. Therefore, AC-7 introduces explicitly the notion of **latest check value** by the data **last** associated with every value. AC-7 ensures the property: If last$[(x, a)] = (y, b)$ then there is no support (y, a) in $D(y)$ with $a < b$. If no support is found by inference, then AC-7 uses an improvement of the AC-6's method to find a support in $D(y)$. When we want to know whether (y, b) is a support of (x, a), we can immediately give a negative answer if last$[(y, b)] > (x, a)$, because in this case we know that (x, a) is a not a support of (y, b) and so that (y, b) is not a support for (x, a). The properties on which AC-7 is based are often called bidirectionnalities. Hence, AC-7 is able to save some checks in the domain in regards to AC-6, while keeping the same space and time complexity.

AC-Inference: This algorithm uses the S-lists of AC-6 to determine in the same way the values for which a support must be sought, but the search for a new support is different from the AC-6's method. For every value (x, a), two lists of values are used: **P-list**$[(x, a)]$ and **U-list**$[(x, a)]$. P-list$[(x, a)]$ contains some supports of (x, a), where as U-list$[(x, a)]$ contains the values for which their compatibility with (x, a) has never been tested. When a support is sought for (x, a), it checks first if there is a valid value in P-list$[(x, a)]$, and every non valid value that is reached is removed from the P-list. If there is no valid value is found in the P-list, then the values of U-list$[(x, a)]$ are successively considered until a valid support is found. Every value of the U-list$[(x, a)]$ which is checked is removed from the U-list. When a new support is found, then some inference rules can be applied to deduce new supports and the U-list and P-list are accordingly modified. For instance if (x, a) is found to be a support for (y, b) then it is inferred that (y, b) is a support for (x, a). Some other inference rules can be used like

commutativity or reflexivity (see in [2].) The space and time complexities are in $O(d^2)$ per constraint.
AC-Identical: This is an extension of AC-Inference which exploits the fact that the same constraint can be defined on different variables. In this case, any knowledge obtain from one constraint is inferred for the other similar constraints.
AC-2000: This is a modification of AC-3. The pending values are the values of $D(x)$ that have a support in $\Delta(y)$. No extra data is used, so it is costly to compute the pending values. Thus, AC-2000 proposes to use this set of pending values only if $|\Delta(y)| < 0.2|D(x)|$; otherwise $D(x)$ is considered as in AC-3. Therefore, AC-2000 is the first adaptive AC algorithm.
AC-2001: This algorithm is based on AC-3 and uses the last data of AC-6. That is, the pending values are the same as for AC-3 and function existValidSupport is similar as the AC-6'one, except that it is checked if the last value is valid. This algorithm inherits of the space complexity of AC-6, without using the S-lists. Note also that this presentation of AC-2001 is original and simpler than the one given in [3].
AC-3.3: AC-3.3 is an improvement of AC-2001 which associates with every value (x, a) a counter corresponding to a lower bound of the size of S-list$[(x, a)]$. The algorithm does not use any S-list, but counters instead. When a support is sought for, the counter of (x, a) is first tested, if it is strictly greater than 0 then we know that a valid support exists. This support cannot be identified but we know that there is one. If (y, b) is deleted then the counters of all the values supported by (y, b) are decremented.

We will not consider AC-8 [4], AC-3_d [8], and AC-3.2 [5], because they mainly improve AC-3 by proposing to propagate the constraint in regards to specific orderings, and this is not our purpose.

The AC algorithms may use the following data structures:
Support: the current support of (x, a) is denoted by support$[(x, a)]$.
Last: the last value of (x, a) for a constraint C is represented by last$[(x, a)]$ which is equals to a value of y or nil.
S-List, P-List, U-list: these are classical list data structures. For any list L we will consider that functions add$(L, (x, a))$ and remove$(L, (x, a))$ are available. These functions respectively add (x, a) to L, and remove (x, a) from L. We will also assume that these function and the size of a list can be computed in $O(1)$.
Tuple counters: there are represented by counter$[(x, a)]$ which counts the number of tuples in $T(C)$ containing (x, a) that are valid.
Table: A table is the set of tuples $T(C)$ associated with two functions: firstTuple(C, y, b) which returns the first tuple of $T(C)$ containing (y, b) nextTuple$(C, x, y, b, a))$ which returns the first tuple of $T(C)$ containing (y, b) and following the tuple $((x, a), (y, b)$. These functions return nil when no such specified tuple exists.

Now, we propose to identify the different concepts used by the existing algorithms instead of having one function per algorithm and one parameter corresponding to each specific algorithm.

Algorithm 3: Pending values selection depending on $pvMode$ (b is a local data.)

$pvMode =$ pvD

firstPendingValue$(C, x, y, \Delta(y))$: return first$(D(x))$

nextPendingValue$(C, x, y, \Delta(y), a)$: return next$(D(x), a)$

$pvMode =$ pvΔs

firstPendingValue$(C, x, y, \Delta(y))$: value
- $b \leftarrow$first$(\Delta(y))$
- return traverseS-list$(C, x, y, \Delta(y))$

nextPendingValue$(C, x, y, \Delta(y), a)$: value
- return traverseS-list$(C, x, y, \Delta(y))$

traverseS-list$(C, x, y, \Delta(y))$: value
- **while** $(y, b) \neq nil$ **do**
  - $(x, a) \leftarrow$seekValidSupportedValue(C, y, b)
  - **if** $(x, a) \neq nil$ **then** return (x, a)
  - $b \leftarrow$next$(\Delta(y), b)$
- return nil

$pvMode =$ pvΔt

firstPendingValue$(C, x, y, \Delta(y))$: value
- $b \leftarrow$first$(\Delta(y))$
- $(x, a) \leftarrow$firstTuple(C, y, b)
- return traverseTuple$(C, x, y, \Delta(y), a)$

nextPendingValue$(C, x, y, \Delta(y), a)$: value
- $a \leftarrow$nextTuple(C, y, b, a)
- return traverseTuple$(C, x, y, \Delta(y), a)$

traverseTuple$(C, x, y, \Delta(y), a)$: C-value
- **while** $(y, b) \neq nil$ **do**
  - **while** $(x, a) \neq nil$ **do**
    - **if** $a \in D(x)$ **then** return (x, a)
    - $(x, a) \leftarrow$nextTuple(C, x, y, b, a)
  - $b \leftarrow$next$(\Delta(y), \delta a)$
  - $(x, a) \leftarrow$firstTuple$(C, x, y, \delta a)$
- return nil

$pvMode =$ pvΔc

firstPendingValue$(C, y, \Delta(y))$: value
- $a \leftarrow$first$(D(x))$
- return seekCompatible$(C, x, y, \Delta(y), a)$

nextPendingValue$(C, y, \Delta(y), a)$: value
- $a \leftarrow$next$(D(x), a)$
- return seekCompatible$(C, x, y, \Delta(y), a)$

seekCompatible$(C, x, y, \Delta(y), a)$: value
- **while** $(x, a) \neq nil$ **do**
  - **for** *each* $b \in \Delta(y)$ **do**
    - **if** $((x, a), (y, b)) \in T(C)$ **then** return (x, a)
  - $(x, a) \leftarrow$next$(D(x), a)$
- return nil

$pvMode =$ pvG: example of generic function: $<$ constraint

firstPendingValue$(C, y, \Delta(y))$: value
- return next$(D(x), \max(D(y)) - 1)$

nextPendingValue$(C, y, \Delta(y), a)$: return next$(D(x), a)$

Thus, we will have a configurable algorithm from which every AC algorithm could be obtain by combining some parameters, each of them corresponding to a concept.

4 Pending values

Finding efficiently a small set of pending values is difficult because pending values sets deal with two different notions at the same time: validity and support. Thus, several set of pending values have been considered. We identify four sets:

1. The values of $D(x)$ (like in AC-3, AC-2001, AC-3.3). This set is denoted by $\underline{\mathsf{pvD}}$.
2. The valid values currently supported by the values of $\Delta(y)$, that is the valid values in the S-lists of $\Delta(y)$. This set is used by AC-6, AC-7, AC-Inference, AC-Identical. It is denoted by $\underline{\mathsf{pv}\Delta\mathsf{s}}$.
3. The values that belong to a tuple containing a value of $\Delta(y)$. A value is pending as many times as it is contained in such a tuple. AC-4 uses this set, and it is denoted by $\underline{\mathsf{pv}\Delta\mathsf{t}}$.
4. The values of $D(x)$ compatible with at least one value of $\Delta(y)$, as in AC-2000. This set is denoted by $\underline{\mathsf{pv}\Delta\mathsf{c}}$.

Since we aim to have a generic algorithm, we propose to define a fifth type: $\underline{\mathsf{pvG}}$ which represents any function given by the user. For instance, for $x < y$ only the modifications of the maximum value of $D(y)$ can lead to new deletions. Thus, the pending values are the values of $D(x)$ that are greater than the maximum value of $D(y)$.

Algorithm 3 is a possible implementation of the computation of pending values. Depending on the set of pending values, the algorithm traverses a particular set. Note that some functions require "internal data" (a data whose value is stored). We assume that FIRST$(D(x))$ returns the first value of $D(x)$ and NEXT$(D(x), a)$ returns the first value of $D(x)$ strictly greater than a. Function SEEKVALIDSUPPORTEDVALUE(C, x, a) returns a valid supported value belonging to the S-list$[(y, b)]$.

Algorithm 4: Function SEEKVALIDSUPPORTEDVALUE

```
SEEKVALIDSUPPORTEDVALUE(C, x, a) : value
    for each value (y, b) ∈ S-list[(x, a)] do
        if b ∈ D(y) then return (y, b)
        REMOVE(S-list[(x, a)], y, b)
    return nil
```

Algorithm 5: Function existValidSupport

```
existValidSupport(C, x, a, y, sMode): boolean
    if slist then remove(S-list[support[(x, a)]], (x, a))
    (y, b) ← nil
    if last then if last[(x, a)] ∈ D(y) then (y, b) ← last[(x, a)])
    if inf and (y, b) = nil then
        (y, b) ← seekValidSupportedValue(C, x, a)
    if (y, b) = nil then (y, b) ← sMode.seekSupport(C, x, a, y)
    if slist then if (y, b) ≠ nil then add(S-list[(y, b)], (x, a))
    if (y, b) ≠ nil then support[(x, a)] ← (y, b)
    return ((y, b) ≠ nil)
```

5 Existence of a valid support

This function also differentiates the existing algorithms. Almost each algorithm uses a different method. We can identify eight ways to determine whether a support exists for (x, a):

1. Check in the domain from scratch (AC-3, AC-2000.)
2. Check if the last value is still valid and if not check in the domain from the last value (AC-2001.)
3. Check in the domain from the last value (AC-6.)
4. Test if a support can be found in S-list$[(x, a)]$, then check in the domain from the last value. Wen searching in the domain uses the fact that last values are available to avoid explicit compatibility checks (AC-7.)
5. Check if there is a valid support in P-list$[(x, a)]$, if there is none check the compatibility with the valid values of U-list$[(x, a)]$. When some compatibility checks are made, then deduce the results of some other compatibility checks and update accordingly some U-lists and P-lists (AC-Inference, AC-identical.)
6. Decrement the counter storing the number of valid supports and test if is strictly greater than 0 (AC-4.)
7. A specific function dedicated to the constraint is used.
8. Use of counters storing a lower bound of the size of the S-list of (x, a), and check in the domain from the last value (AC-3.3.)

The last point deserves a particular attention. The time complexity of using a counter of the number of elements in a list is the same as the management of the list. Moreover, AC-3.3 implies that the counters are immediately updated when a value is removed, which is not the case with AC-7, and the lazy approach used by AC-7 to maintain the consistency of the S-list has been proved more efficient. Therefore, we will prefer the explicit use of S-lists to the use of a lower bound of the size of the S-lists of AC-3.3.

We propose to consider the following parameters:

• last: the search for a valid support is restarting from the last value. The last value is also used to avoid some negative checks (AC-6, AC-7, AC-Inference, AC-Identical, AC-2001, AC-3.3.)

Algorithm 6: Functions seeking for a valid support

```
                                        sMode = sD
seekSupport(C, x, a, y) : value
    b ← first(D(y))
    if last then
        b ←next(D(y),last[(x, a)])
        while b ≠ nil do
            if last[(y, b)] ≤ (x, a) and ((x, a), (y, b)) ∈ T(C) then
                last[(x, a)] ← (y, b)
                return (y, b)
            b ←next(D(y), b)
    else
        while b ≠ nil do
            if ((x, a), (y, b)) ∈ T(C) then return (y, b)
            b ←next(D(y), b)
    return nil
                                        sMode = sC
seekSupport(C, x, a, y) : value
    counter[(x, a)] ← counter[(x, a)] − 1
    if counter[(x, a)] = 0 then return nil
    else return (y,first(D(y)))
                                        sMode = sT
seekSupport(C, x, a, y) : value
    for each (y, b) ∈ P-list[(x, a)] do
        remove(P-list[(x, a)], (y, b))
        if b ∈ D(y) then return (y, b)
    for each (y, b) ∈ U-list[(x, a)] do
        remove(U-list[(x, a)], (y, b))
        remove(U-list[(y, b)], (x, a))
        if ((x, a), (y, b)) ∈ T(C) then
            add(P-list[(y, b)], (x, a))
            if b ∈ D(y) then return (y, b)
    return nil
                sMode = sGen example of generic function: < constraint
seekSupport(C, x, a, y) : return false
```

• inf: the search for a valid support is first done by searching for a valid supported value in the S-list (AC-7, AC-3.3.)

• slist: this parameter means that the S-lists are used. It is implies by inf and pvΔs (AC-6, AC-7, AC-Inference, AC-Identical, AC-3.3.)

• sD: the search for a support is made by testing the compatibility between (x, a) and the values in $D(y)$ (AC-3, AC-6, AC-7, AC-2000, AC-2001, AC-3.3.)

• sC: the search for a valid support consists of decrementing the counter of valid tuples and checking if it is > 0 (AC-4.)

• sT: the search for a valid support is made by testing the validity of the values of P-list$[(x, a)]$ and by checking if there is a value in U-list$[(x, a)]$ which is valid and compatible with (x, a) (AC-Inference, AC-Identical.)

• sGen: the search for a valid support is defined by a function provided by the user and dedicated to the constraint. We present an example for the $<$ constraint.

From these parameters we can now propose a possible code for Function EXISTVALIDSUPPORT of CAC algorithm (see Algorithm 5.) Possible instantiations of Function SEEKSUPPORT are given by Algorithm 6. An example is also given for constraint $<$.

6 Analysis of different methods

The main issue of AC algorithms is to deal with two different notions: support and validity. It is difficult to handle these two notions at the same time. Thus, the algorithms usually privilegiate one notion:

• When constructing the pending values set, pvD algorithms totally ignores the notion of support. The other algorithms try to combine the two notions: pvDc algorithms consider first the validity, whereas pvΔt algorithms deal first with all supports. And, pvΔs algorithms traverse the current supported values and check the validity.

• When searching for a new support, sD algorithms consider the valid values, and the check if there are support, whereas sT algorithms traverse the supports and check for their validity.

7 Nomenclature

From the different concepts we have identified we can propose a new nomenclature for the AC algorithms. Until now, the naming used the prefix "AC-" followed by a number or date. Excepted AC-Inference or AC-Identical which have tried to express a little bit some ideas of the algorithms, it is clearly impossible to understand the specificity of each algorithm from their name.

The nomenclature we propose uses CAC as prefix which stands for Configurable Arc Consistency algorithm. Then the combinations of parameters corresponding to the AC algorithm are added to the CAC prefix. For instance, CAC-pvD-sD means that the pending values are the values of $D(x)$ and that a

new support is sought in the domain by checking the compatibilities between values. This is exactly the description of AC-3. For adaptive algorithm a "/" is used to differentiate the possibilities: AC-2000 is renamed CAC-pvΔc/pvD-sD. We can describe all the existing algorithms:

name	new name
AC-3	CAC-pvD-sD
AC-4	CAC-pvΔt-sC
AC-6	CAC-pvΔs-last-sD
AC-7	CAC-pvΔs-last-inf-sD
AC-Inference or AC-Identical	CAC-pvΔs-sT
AC-2000	CAC-pvΔc/pvD-sD
AC-2001	CAC-pvD-last-sD
AC-3.3	CAC-pvD-last-inf-sD

Note that CAC-pvΔs-last-sD is a slight improvement of AC-6, because some negative checks are avoided.

8 Adaptive algorithm

The advantage of adaptive algorithm is to avoid some pathological cases of each algorithm. A property which exactly differentiates AC-2001 and AC-6 in regards to the number of operations they need to establish arc consistency has been given in [3]:

Property 2 *The number of values that are considered to find the pending values in:*

- *a* $\underline{\text{pvD}}$ *oriented algorithm is* $\#(pvD) = |D(x)|$.
- *a* $\underline{\text{pv}\Delta\text{s}}$ *oriented algorithm is*
$$\#(pv\Delta s) = |\Delta(y)| + \sum_{b \in \Delta(y)} |S\text{-}list[(y,b)]|$$

These two numbers are sufficient to differentiate AC-2001 and AC-6 because they use both the same algorithm to find a support for a value. This is clearly shown by their new names that are respectively CAC-pvD-last-sD and CAC-pvΔs-last-sD. So, by considering the method to find the pending values that studies the smallest number of values we can define an algorithm which is better than any of two previous ones. We can use first a $\underline{\text{pv}\Delta\text{s}}$ oriented algorithm and then switch to a $\underline{\text{pvD}}$ one and conversely.

Unfortunately, it is difficult to quickly compute $\#(pv\Delta s)$. The sum, indeed, needs to consider every value of the delta domain independently. However, we immediately have: $\#(pv\Delta s) \geq 2|\Delta(y)|$, and $|\Delta(y)|$ can be incrementally maintained, thus we can consider that we know its value in $O(1)$. Algorithm 7 is a possible implementation of the functions selecting $sMode$ and $pvMode$ that is used by Algorithm 2 (AC-2000 is taken into account in this function.) Switching from a type of algorithm to another one can also cause some other problems, because the different types of algorithms do not use the same data structures. When switching from an algorithm using a data structure to an algorithm that

Algorithm 7: Selection of pvMode and sMode

```
selectPendingValueMode(C,x,y,Δ(y),pvMode):pvMode
    if pvMode=pvΔc/pvD then
        if |Δ(y)| < 0.2|D(x)| then return pvΔc
        return pvD
    if pvMode=pvD/pvΔs then
        if |D(x)| < 2.|Δ(y)| then return pvD
        if |D(x)|<|Δ(y)|+∑_{b∈Δ(y)} |S-list[(y,b)]| then return pvD
        return pvΔs
    return pvMode
selectExistSupportMode(C, x, a, sMode) : sMode
    if sMode=sD/sT then
        if |D(x)| < |P-list[(x,a)]| then return sD
        return sT
    return sMode
```

does not use that data structure we have two possibilities: either the data structure is updated after switching, or it is systematically updated even if it is not used. For the S-lists, the cost to maintain them is $O(1)$ per deletion or addition therefore the second solution is simpler. For the U-lists and the P-lists there is no problem because they do not need to be updated.

We have seen that it is possible to change the way the pending values are computed. Two other possibilities are: use ot not the inf parameter, and switch from sD and sT and conversely. However, it is much more complicated to find a good criteria of selection, because the lists are modified when using inf or sT. Thus, at a given moment the size of the list can be not in favor of one method but becomes strongly in favor of this method after its application. After some experiments it appears that the switch from inf to no inf does not change anything, and it appears that it is interesting to switch from sD to sT and conversely. If the size of the domain is smaller than the size of the P-list then sD is selected, else sT is selected. (see Algorithm 7.)

9 Experiments

We propose a comparisons of the MAC version of the algorithms on the well-known RLFAPs benchmarks. We give the results only for instances SCEN#1, SCEN#11, SCEN#8 because the results are quite representative (and also due to the lack of space). No specific ordering are consider for the constraints (all the algorithms consider the same ordering). For each algorithm we give the ratio between the time needed by the algorithm to solve the problem over the time needed by the best algorithm:

pvΔt	•											
pvΔc			•									
pvΔs						•	•	•	•	•	•	•
pvD		•	•	•	•			•	•		•	•
last				•	•	•	•	•	•		•	•
inf					•		•		•			
sD		•	•	•	•	•	•	•	•			•
sC	•											
sT										•	•	•
#1	19.4	4.2	3.7	2.8	2.4	3.7	3.7	3.2	3.2	1.5	1.1	1
#11	15	7	6.8	4	3.3	2.6	2.5	1.8	1.8	1.7	1.1	1
#8	59.3	68.2	67.3	50	48	21	19	4	3	4	1.1	1

This results show clearly that the adaptive algorithm performs better than non adaptive and that the $\underline{\text{sT}}$ algorithms are better than the others.

10 Conclusion

We have presented CAC a new configurable, generic and adaptive algorithm, which is able to represent all existing algorithms. We have clearly differentiate all the existing algorithms thanks to the identification a the most important concepts. We have proposed new combination of concepts that perform well in practice as shown by the experimental results we gave.

References

1. C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 1994.
2. C. Bessière, E. Freuder, and J.-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.
3. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle, WA, USA, 2001.
4. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):79–89, 1998.
5. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionnality in coarse-grained arc consistency algorithm. In *Proceedings CP'03*, pages 480–494, Cork, Ireland, 2003.
6. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
7. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
8. M. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings CP'02*, pages 755–760, Ithaca, NY, USA, 2002.
9. P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
10. Y. Zhang and R. Yap. Making ac-3 an optimal algorithm. In *Proceedings of IJCAI'01*, pages 316–321, Seattle, WA, USA, 2001.

Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems

Xuan-Ha Vu[1], Hermann Schichl[2], and Djamila Sam-Haroud[1]

[1] Artificial Intelligence Laboratory,
Swiss Federal Institute of Technology in Lausanne (EPFL),
CH-1015, Lausanne, Switzerland
{xuan-ha.vu, jamila.sam}@epfl.ch
http://liawww.epfl.ch
[2] Faculty of Mathematics,
University of Vienna,
Nordbergstr. 15, A-1090 Wien, Austria
hermann.schichl@esi.ac.at
http://www.mat.univie.ac.at/ herman

Abstract. The pioneering paper H. Schichl and A. Neumaier [1] has founded the fundamentals of interval analysis on DAGs for global optimization, including a fundamental of constraint propagation. In this paper, we extend the constraint propagation technique for solving numerical constraint satisfaction problems. In particular, we propose an advanced constraint propagation technique, which makes the constraint propagation practical and efficient, and a method to coordinate constraint propagation and exhaustive search, which uses a single DAG for each problem. The experiments carried out on various problems show that the new approach outperforms previously available propagation techniques by an order of magnitude or more in speed, while being roughly the same in quality measures.

1 Introduction

Many real-world problems require solving numerical constraint satisfaction problems (NCSPs). An NCSP is a triplet $(\mathcal{V}, \mathcal{C}, \mathcal{D})$ which consists of a finite set $\mathcal{V}$ of variables taking their values in domains $\mathcal{D}$ over the reals and subject to a finite set $\mathcal{C}$ of *numerical* constraints. A tuple of values assigned to the variables such that all the constraints are satisfied is called a solution. The set of all the solutions is called the solution set. In practice, numerical constraints are often equalities or inequalities expressed in *factorable* form, that is, they can be represented by elementary functions such as $+$, $-$, $\times$, $\div$, log, exp, sin, cos,... In other words, such an NCSP can be expressed as follows

$$F(x) \in \mathbf{b},\ x \in \mathbf{x} \tag{1}$$

where $F : \mathbb{R}^n \to \mathbb{R}^m$ is a factorable function, x is a vector of n real variables, $\mathbf{x}$ and $\mathbf{b}$ are interval vectors of size n and m respectively.

Many solution techniques have been proposed in *Constraint Programming* and *Mathematical Programming* to solve NCSPs. To achieve full rigor when dealing with floating-point numbers, most solution techniques have been based on *interval arithmetic* or its variants. In the last ten years, there have been elaborate uses of interval arithmetic to devise the notions of *inclusion test* and *contractor* as described in the book JAULIN *et al.* [2]. An inclusion test is to check the inclusion of variable domains in the solution set. A contractor, possible variants are *narrowing operator*s [3, 4] and *contracting operator*s [5–7], is a method to narrow the variable domains such that no solution is lost. Various basic inclusion tests and contractors have been described in [2]. Recently, there has been a new approach, called *interval constraint propagation*, which associates *constraint propagation/local consistency* techniques in artificial intelligence with interval analytic methods to devise advanced contractors, e.g., the so-called *forward-backward contractor* [4, 2]. A representation of the solutions can be often computed by interleaving inclusion tests or contractors with *exhaustive search*; the solution techniques often use *bisection* search to solve the problems exhaustively. However, advanced search techniques (see SILAGHI *et al.* [6] and VU *et al.* [7]) have also been proposed to improve the search performance for problems with a continuum of solutions (e.g., inequalities), while maintaining the same performance for problems with isolated solutions (e.g., equalities).

Most recently, a fundamental framework for interval analysis on DAGs has been proposed by SCHICHL & NEUMAIER [1], which includes an extension of forward-backward propagation for working on DAGs. In order to exploit the framework and make it useful for more applications, in this paper we extend the DAG-based constraint propagation technique for solving NCSPs. In Section 3, we describe the DAG representation of problems and new extensions of the forward evaluation and the backward propagation. In Section 4, we propose an advanced constraint propagation technique, which makes the above framework for constraint propagation efficient and practical, and a method to coordinate constraint propagation and exhaustive search using a single DAG for each problem. Finally, as one can see in Section 5, the experiments carried out on various problems show that the new approach outperforms previously available propagation techniques by an order of magnitude or more in speed, while being roughly the same in quality measures.

2 Background

2.1 Interval Arithmetic

Interval arithmetic is an extension of real arithmetic defined on the set of real intervals, rather than the set of real numbers. According to KEARFOTT [8], a form of interval arithmetic perhaps first appeared in 1924 in BURKILL [9]. Modern development of interval arithmetic began with R. E. MOORE's dissertation [10]. Fundamentally, if $\mathbf{x}$ and $\mathbf{y}$ are two real intervals, then the four elementary operations for *idealized interval arithmetic* obey the rule: $\mathbf{x} \diamond \mathbf{y} = \{x \diamond y \mid x \in \mathbf{x}, y \in \mathbf{y}\}, \forall \diamond \in \{+, -, \times, \div\}$. Thus, the ranges of the four elementary interval

arithmetic operations are exactly the ranges of the their real-valued counterparts. Although this rule characterizes these operations mathematically, interval arithmetic's usefulness is due to the *operational definitions* based on interval bounds [11]. For example, let $\mathbf{x} = [\underline{x}, \overline{x}]$ and $\mathbf{y} = [\underline{y}, \overline{y}]$, we define

$$\begin{aligned}
&\mathbf{x} + \mathbf{y} = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
&\mathbf{x} - \mathbf{y} = [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
&\mathbf{x} \times \mathbf{y} = [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\
&\mathbf{x} \div \mathbf{y} = \mathbf{x} \times 1/\mathbf{y} \text{ if } 0 \notin \mathbf{y}, \text{ where } 1/\mathbf{y} = [1/\overline{y}, 1/\underline{y}]
\end{aligned}$$

Moreover, if such operations are composed, *bounds on the ranges* of factorable real functions can be obtained.

The finite nature of computers precludes an exact representation of the *reals*. In practice, the real set, $\mathbb{R}$, is approximated by a finite set $\mathbb{F}_\infty = \mathbb{F} \cup \{-\infty, +\infty\}$, where $\mathbb{F}$ is the set of floating-point numbers. The set of real intervals is then approximated by the set $\mathbb{I}$ of intervals with bounds in $\mathbb{F}_\infty$. The power of interval arithmetic lies in its implementation on computers. In particular, *outwardly rounded* interval arithmetic allows *rigorous enclosures* for the ranges of operations and functions. This makes a qualitative difference in scientific computations, since the results are now intervals in which the exact result must lie. Readers are referred to [12, 11, 2] for more details on basic interval methods.

2.2 Interval Constraint Propagation

The *tree representation* of constraint systems has been proposed in Benhamou *et al.* [4], therein each factorable constraint $r(t_1, \ldots, t_k)$ is represented by an *attribute tree* whose root node represents the k-ary relation symbol r, and the terms t_i are composed of nodes representing either a variable, a constant, or an elementary operation. Moreover, each node but the root is associated with two intervals, one for forward evaluation and the other for backward propagation.

The constraint propagation algorithm `HC4` in [4], also referred to as the forward-backward contractor (see [2]), is based on the following two main processes. The first one is the *forward evaluation* which is recursively performed by a post-order traversal of the tree representation from leaves to roots in order to evaluate the ranges of sub-expressions represented by the tree nodes using *natural interval extension*. The second one is the *backward propagation* on the tree representation which is recursively performed by a pre-order traversal of the tree representation of each constraint from root to leaves in order to prune the corresponding interval associated with each node of the tree using the *projection narrowing operator* associated with the father of the node. Readers are referred to [4] for more details.

3 Numerical Constraint Propagation on DAGs

We will consider a constraint system of the form (1); the constraints can be equations or inequalities depending on whether the corresponding components of $\mathbf{b}$, called *constraint ranges*, are thin intervals (i.e. of the form $[b_i, b_i]$).

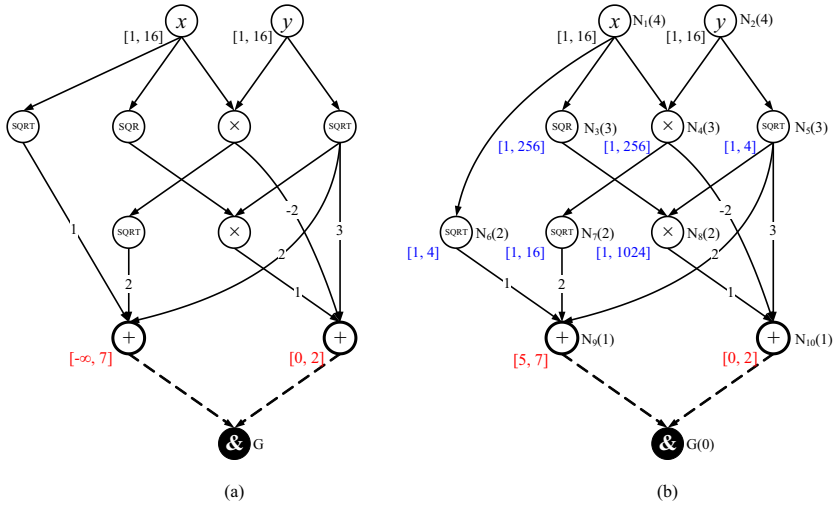


Fig. 1. The DAG representation (a) before and (b) after performing node ordering and recursive forward evaluation

Example 1. Consider the following parametric constraint system

$$\begin{cases} \sqrt{x} + 2\sqrt{xy} + 2\sqrt{y} \leq 7, \\ x^2\sqrt{y} - 2xy + 3\sqrt{y} \in [p, q], \\ x \in [1, 16],\ y \in [1, 16]. \end{cases} \tag{2}$$

The first constraint is an inequality with constraint range $[-\infty, 7]$. The second constraint can be either an equation or an inequality depending on the parameters (p, q). For instance, the second constraint is an equation if $(p, q) = (0, 0)$ and an inequality if $(p, q) = (0, 2)$. Throughout this paper, we will use $(p, q) = (0, 2)$.

3.1 DAG Representation

We assume that readers are already familiar with fundamental concepts in graph theory like *directed acyclic graph/multigraph*. In the representation of the NCSPs we will use *directed acyclic multigraph with ordered edges* (for the definition readers are referred to [1] and references therein); for short, this is a directed acyclic multigraph, in which the incoming and outgoing edges at every node are totally ordered.

Theorem 1. *For every directed acyclic multigraph (V, E, f) there exists a total order $\preceq$ on the vertices V such that $\forall v \in V$: if u is an ancestor of v, then $v \preceq u$.*

We use a directed acyclic multigraph, whose edges are totally ordered, together with an ordering on the vertices, as obtained in Theorem 1, to represent the constraint system (1), for short we call it a *Directed Acyclic Graph* (DAG). In that *DAG representation*, every node represents an elementary operation such

as $+$, $\times$, $\div$, log, exp, ... and every edge represents the computational flow associated with a coefficient. In practice, we have to use multigraphs instead of simple graphs for the representation because some special operations can take the same input more than once, for example, when the expression x^x is represented by the elementary power operation x^y. The ordering of edges is needed for non-commutative operations like the division, but not for commutative operations. For convenience, a virtual ground node is added to the DAG to be the parent of all the nodes representing the constraints. In fact, the ground node can be interpreted as the logical 'AND' operation. Each node $\mathbf{N}$ in the DAG is associated with an interval, denoted $\mathbb{I}(\mathbf{N})$, in which the exact range of the corresponding sub-expression must lie.

Example 2. The DAG representation of (2) is depicted in Figure 1. The sequence of nodes $\{\mathbf{N}_1, \mathbf{N}_2, \ldots, \mathbf{N}_{10}\}$ is an ordering of the nodes that satisfies Theorem 1.

3.2 Forward Evaluation and Backward Propagation on DAGs

In practice, we often see functions of the form $f : D \to \mathbb{R}^m$, where $D \subset \mathbb{R}^n$. Quite often, in range analysis we need f to accept input from the domain $\mathbb{R}^n$, so we have to find a proper way to extend functions in a consistent way.

Definition 1 (Extended Function). *Let $f : D \to \mathbb{R}^m$ be a function, where $D \subseteq \mathbb{R}^n$, and S a subset of $2^{\mathbb{R}}$, the power set of $\mathbb{R}$. A function $g : \mathbb{R}^n \to \mathbb{R}^m \cup S^m$ is called an* S-extended function *of f if*

$$g(x) = \begin{cases} f(x) & \text{if } x \in D, \\ y \in S^m & \text{otherwise} \end{cases} \tag{3}$$

It is to easy see that there is only one S-extended function if S has only one element, for instance, when S is either $\{\emptyset\}$ or $\{\mathbb{R}\}$.

Example 3. The domain of the standard division x/y is $D_{\div} = \{(x, y) \in \mathbb{R}^2 \mid y \neq 0\}$. The unique $\{\emptyset\}$-extended function of the standard division is defined by

$$x \div_{\emptyset} y = \begin{cases} x/y & \text{if } y \neq 0, \\ \emptyset & \text{otherwise} \end{cases} \tag{4}$$

The unique $\{\mathbb{R}\}$-extended function of the standard division is defined by

$$x \div_{\mathbb{R}} y = \begin{cases} x/y & \text{if } y \neq 0, \\ \mathbb{R} & \text{otherwise} \end{cases} \tag{5}$$

The following is a $\{\emptyset, \mathbb{R}\}$-extended function of the standard division:

$$x \div_{\star} y = \begin{cases} x/y & \text{if } y \neq 0, \\ \emptyset & \text{if } x \neq 0, y = 0, \\ \mathbb{R} & \text{otherwise} \end{cases} \tag{6}$$

In the next definition, we extend the concept of inclusion function of [2].

Definition 2 (Inclusion Function). *Let S be a subset of $2^{\mathbb{R}}$, and $f : \mathbb{R}^n \to \mathbb{R}^m \cup S^m$ an S-extended function of a function $\varphi : D \to \mathbb{R}^m$, where $D \subseteq \mathbb{R}^n$. A function $[f] : \mathbb{I}^n \to \mathbb{I}^m$ is called an* inclusion function *of f and of φ if $\forall \mathbf{x} \in \mathbb{I}^n$: $\underline{f(\mathbf{x}) \subseteq [f](\mathbf{x})}$, where $f(\mathbf{x}) \equiv \{f(x) \mid x \in \mathbf{x} \cap D\} \cup \bigcup_{x \in \mathbf{x} \setminus D} f(x)$.*[3]

[3] The set union of vectors is performed in component-wise fashion.

Example 4. Let $\mathbf{x} = [\underline{x}, \overline{x}], \mathbf{y} = [\underline{y}, \overline{y}]$. We give as example three natural inclusion functions for the divisions defined by (4), (5) and (6) respectively.

$$\mathbf{x}[\div_{\emptyset}]\mathbf{y} = \begin{cases} \emptyset & \text{if } \mathbf{y} = [0,0], \\ [0,0] & \text{else if } \mathbf{x} = [0,0], \\ \mathbf{x} \div \mathbf{y} & \text{else if } 0 \notin \mathbf{y}, \\ [\underline{x}/\overline{y}, +\infty] & \text{else if } \underline{x} \geq 0 \wedge \underline{y} = 0, \\ [-\infty, \underline{x}/\underline{y}] & \text{else if } \underline{x} \geq 0 \wedge \overline{y} = 0, \\ [-\infty, \overline{x}/\overline{y}] & \text{else if } \overline{x} \leq 0 \wedge \underline{y} = 0, \\ [\overline{x}/\underline{y}, +\infty] & \text{else if } \overline{x} \leq 0 \wedge \overline{y} = 0, \\ [-\infty, +\infty] & \text{otherwise} \end{cases} \tag{7}$$

$$\mathbf{x}[\div_{\mathbb{R}}]\mathbf{y} = \begin{cases} \mathbf{x} \div \mathbf{y} & \text{if } 0 \notin \mathbf{y}, \\ [-\infty, +\infty] & \text{otherwise} \end{cases} \tag{8}$$

$$\mathbf{x}[\div_{\star}]\mathbf{y} = \begin{cases} \mathbf{x}[\div_{\emptyset}]\mathbf{y} & \text{else if } 0 \notin \mathbf{x} \vee 0 \notin \mathbf{y}, \\ [-\infty, +\infty] & \text{otherwise} \end{cases} \tag{9}$$

It is easy to see that $\forall \mathbf{x}, \mathbf{y} \in \mathbb{I} : \mathbf{x}[\div_{\emptyset}]\mathbf{y} \subseteq \mathbf{x}[\div_{\star}]\mathbf{y} \subseteq \mathbf{x}[\div_{\mathbb{R}}]\mathbf{y}$. Unfortunately, some interval implementations use the division $[\div_{\mathbb{R}}]$, while it is safe to use the division $[\div_{\emptyset}]$ in some computations such as forward evaluation, as described hereafter.

The *natural inclusion function* of f (see [2]), denoted by $\mathbf{f}$, is an example of an inclusion function: in the factorable form of f each real variable is replaced by an interval variable and each operation is replaced by its interval counterpart.

In the DAG representation of (1), let $\mathbf{N}$ be a node which is not the ground node and has k children $\{\mathbf{C}_i\}_{i=1}^k$. The elementary operation represented by $\mathbf{N}$ is a function $f : D_f \to \mathbb{R}$, where $D_f \subseteq \mathbb{R}^k$. Hence, the relationship between $\mathbf{N}$ and its children can be written as $\mathbf{N} = f(\mathbf{C}_1, \ldots, \mathbf{C}_k)$.[4] Let $[f]$ be an inclusion function of the $\{\emptyset\}$-extended function of f. The *forward evaluation at node* $\mathbf{N}$ *using the inclusion function* $[f]$ is defined as follows

$$\texttt{FE}(\mathbf{N}, [f]) \equiv \{\mathbb{I}(\mathbf{N}) := \mathbb{I}(\mathbf{N}) \cap [f](\mathbb{I}(\mathbf{C}_1), \ldots, \mathbb{I}(\mathbf{C}_k))\} \tag{10}$$

The aim of forward evaluation is to evaluate the ranges of nodes based on their children's ranges.

The aim of the *backward propagation* is to prune the intervals associated with children based on the constraint range of their parent. In other words, for each child $\mathbf{C}_i$ the backward propagation evaluates the i-th projection of the relation $\mathbf{N} = f(\mathbf{C}_1, \ldots, \mathbf{C}_k)$ on the variable represented by $\mathbf{C}_i$. It is then called the i-th backward propagation at $\mathbf{N}$ and denoted by $\texttt{BP}(\mathbf{N}, \mathbf{C}_i)$. For convenience, we define the following sequence as the backward propagation at node $\mathbf{N}$

$$\texttt{BP}(\mathbf{N}) = \{\texttt{BP}(\mathbf{N}, \mathbf{C}_i)\}_{i=1}^k \tag{11}$$

Although the projection of relations is expensive in general, an evaluation of the projection of elementary operations can be obtained at low cost. Indeed, suppose that from the relation $\mathbf{N} = f(\mathbf{C}_1, \ldots, \mathbf{C}_k)$ we can infer an equivalent

[4] Where we abuse the notation of a node for the real variable represented by it.

relation $\mathbf{C}_i = g_i(\mathbf{N}, \{\mathbf{C}_j\}_{j=1;j\neq i}^k)$ for some $i \in \{1,\ldots,k\}$, where g_i is a function from $\mathbb{R}^k$ to $\mathbb{R}$. Let $[g_i]$ be an inclusion function of g_i. The i-th backward propagation can then be obtained as follows

$$\texttt{BP}(\mathbf{N}, \mathbf{C}_i) \equiv \{\mathbb{I}(\mathbf{C}_i) := \mathbb{I}(\mathbf{C}_i) \cap [g_i](\mathbb{I}(\mathbf{N}), \{\mathbb{I}(\mathbf{C}_j)\}_{j=1;j\neq i}^k)\} \tag{12}$$

In case that we cannot infer such a function g_i, more complicated rules to obtain the i-th projection of the relation $\mathbf{N} = f(\mathbf{C}_1,\ldots,\mathbf{C}_k)$ have to be constructed if the cost is low, alternatively the relation can be ignored. Fortunately, we can evaluate those projections for most elementary operations at low cost.

Definition 3. *Let f be the elementary operation represented by $\mathbf{N}$. We will use the notation $\oslash$ to mean that either the division $[\div_\star]$ or the division $[\div_{\mathbb{R}}]$ can be used at the place the notation $\oslash$ appears. The rules for the forward evaluation and the backward propagation are given as follows:*

- *if f is a univariate function such as sqr, sqrt,* $\exp, \log, \ldots$ *we can define*

$$\texttt{FE}(\mathbf{N}, [f]) \equiv \{\mathbb{I}(\mathbf{N}) := \mathbb{I}(\mathbf{N}) \cap [f](\mathbb{I}(\mathbf{C}_1))\}$$
$$\texttt{BP}(\mathbf{N}, \mathbf{C}_1) \equiv \{\mathbb{I}(\mathbf{C}_1) := \mathbb{I}(\mathbf{C}_1) \cap [f^{-1}](\mathbb{I}(\mathbf{N}))\}\ (f^{-1}(.)\ \textit{is the pre-image})$$

- *if f is defined by $f(x_1,\ldots,x_k) = \alpha + \sum_{i=1}^k \alpha_i x_i$, we define*

$$\texttt{FE}(\mathbf{N}, \mathbf{f}) \equiv \{\mathbb{I}(\mathbf{N}) := \mathbb{I}(\mathbf{N}) \cap (\alpha + \sum_{i=1}^k \alpha_i \mathbb{I}(\mathbf{C}_i))\}$$

$$\texttt{BP}(\mathbf{N}, \mathbf{C}_i) \equiv \{\mathbb{I}(\mathbf{C}_i) := \mathbb{I}(\mathbf{C}_i) \cap \frac{1}{\alpha_i}(\mathbb{I}(\mathbf{N}) - \alpha - \sum_{j=1;j\neq i}^k \alpha_j \mathbb{I}(\mathbf{C}_j))\}\ (i = 1, ..., k)$$

- *if f is defined by $f(x_1,\ldots,x_k) = \alpha \prod_{i=1}^k x_i$, we define*

$$\texttt{FE}(\mathbf{N}, \mathbf{f}) \equiv \{\mathbb{I}(\mathbf{N}) := \mathbb{I}(\mathbf{N}) \cap \alpha \prod_{i=1}^k \mathbb{I}(\mathbf{C}_i)\}$$

$$\texttt{BP}(\mathbf{N}, \mathbf{C}_i) \equiv \{\mathbb{I}(\mathbf{C}_i) := \mathbb{I}(\mathbf{C}_i) \cap (\mathbb{I}(\mathbf{N}) \oslash (\alpha \prod_{j=1;j\neq i}^k \mathbb{I}(\mathbf{C}_j)))\}\ (i = 1, \ldots, k)$$

- *if f is defined by $f(x,y) = x/y$, i.e. $k = 2$, we define*

$$\texttt{FE}(\mathbf{N}, \mathbf{f}) \equiv \{\mathbb{I}(\mathbf{N}) := \mathbb{I}(\mathbf{N}) \cap \mathbf{f}(\mathbb{I}(\mathbf{C}_1), \mathbb{I}(\mathbf{C}_2))\},\ \textit{where}\ \mathbf{f} \in \{[\div_\emptyset], [\div_\star], [\div_{\mathbb{R}}]\}$$
$$\texttt{BP}(\mathbf{N}, \mathbf{C}_1) \equiv \{\mathbb{I}(\mathbf{C}_1) := \mathbb{I}(\mathbf{C}_1) \cap (\mathbb{I}(\mathbf{N}) \times \mathbb{I}(\mathbf{C}_2))\}$$
$$\texttt{BP}(\mathbf{N}, \mathbf{C}_2) \equiv \{\mathbb{I}(\mathbf{C}_2) := \mathbb{I}(\mathbf{C}_2) \cap (\mathbb{I}(\mathbf{C}_1) \oslash \mathbb{I}(\mathbf{N}))\}$$

4 Coordinating Constraint Propagation and Search

We focus on solving techniques following the *branch-and-prune* framework, where the solving process is performed by repeatedly interleaving *pruning* techniques, which use local techniques such as constraint propagation to reduce the variable

```
procedure NodeOccurrences(in : N; out : V_oc)
    for each child C of node N do
        V_oc[C] := V_oc[C] + 1;
        NodeOccurrences(C, V_oc);
    end-for
end
```

Fig. 2. If traversing all active constraints, the `NodeOccurrences` procedure counts the number of occurrences of each node in the factorable form of the active constraints

domains, with *branching* techniques, which split a problem into subproblems. Each subproblem constructed in the solving process usually consists of a subset of constraints, hereafter called *active constraints*, which need to be satisfied, and sub-domains of the initial variable domains. Therefore, solving techniques that use the DAG representation need to create such a representation for each subproblem. The simplest way is to build a new DAG to represent each subproblem. However, the total cost of creating such DAGs for the whole solving process is probably high. As an alternative, we propose in Section 4.1 to modify a piece of information attached to the initial DAG in order to make the initial DAG interpreted as the DAG representation of a subproblem without the necessity of creating new DAGs.

4.1 Partial Forward-Backward Propagation on DAGs

Partial DAG Representation. In order to represent the set of active constraints without having to create new DAGs, we use a vector, V_{oc}, whose size is equal to the number of nodes of the DAG representing the initial problem. For each node $\mathbf{N}$ of the DAG, we use the entry $V_{oc}[\mathbf{N}]$ to count the number of occurrences of $\mathbf{N}$ in the factorable form of the active constraints. In Figure 2, we give a recursive procedure, called `NodeOccurrences`, to compute such a vector. It is easy to see that $V_{oc}[\mathbf{N}] = 0$ if and only if $\mathbf{N}$ is not in the representation of the active constraints. Therefore, by combining the initial DAG with the vector V_{oc}, we have a so-called *partial DAG representation* for each subproblem. In the latter computations, we can use the partial DAG representation in a way similar to using the (full) DAG representation, except that we ignore all nodes corresponding to zeros of the vector V_{oc}. An example of the partial DAG representation for the problem (2) is depicted in Figure 3.

Forward-Backward Propagation on DAGs. Inspired by the original forward evaluation and backward propagation in [4], we devise a new algorithm for numerical constraint propagation, that is based on partial DAG representation instead of tree representation. We call the new algorithm *"Forward-Backward Propagation on a DAG"* and denote it by `FBPD`. In Figure 4, we present the main steps of `FBPD`. In the next paragraphs, we describe in detail the procedures that are not made explicit in Figure 4.

Recursive Forward Evaluation. Similar to the `HC4` algorithm, we perform a recursive forward evaluation at the initialization phase (lines 01-08) to evaluate the

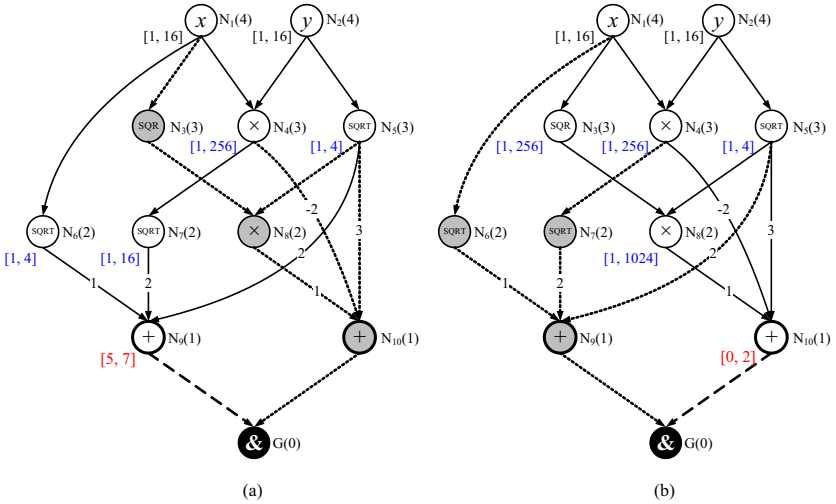


Fig. 3. The partial DAG representation of the problem (2) when (a) the first constraint, or (b) the second constraint is the unique active constraint. The grey nodes are not counted, hence are ignored in computations. The dotted edges are redundant. The node levels are not updated

ranges of the nodes in the partial DAG representation. In Figure 5, we give the details of a procedure, named `ForwardEvaluation`, for such a recursive evaluation. The results of the recursive forward evaluation of (2) are depicted in Figure 1b and Figure 3 for the case that both constraints are active and the case that only one constraint is active, respectively.

Get the Next Node for Further Processing. Like with the `HC4` algorithm [4], in the main body of the `FBPD` algorithm there are two principal processes: forward evaluation and backward propagation. However, unlike the `HC4` algorithm, the `FBPD` algorithm performs these processes for a single node instead of all the nodes at once. Therefore, in the `FBPD` algorithm, the choice of the next node for further processing can be made adaptively based on the results of the previous processes. The algorithm uses two waiting lists to store the nodes waiting for further processing. The first list, $\mathcal{L}_f$, is a list of nodes that is scheduled for forward evaluation, that is, for evaluating its range based on its children's ranges. The second list, $\mathcal{L}_b$, is a list of nodes that is waiting for backward propagation, that is, for pruning its children's ranges based on its range. In general, when $\mathcal{L}_f$ contains many nodes, the nodes should be sorted such that the forward evaluation of a node is performed after the forward evaluation of its children. Analogously, the nodes in $\mathcal{L}_b$ should be sorted such that the backward propagation at a node is performed before the backward propagation at its children. The `NodeLevel` procedure in Figure 6 assigns to each node a *node level* such that the node level of an arbitrary node is smaller than the node levels of its descendants. We then sort the nodes of $\mathcal{L}_b$ and $\mathcal{L}_f$ in ascending order and descending order of node levels, respectively, to meet the above requirements. The `getNextNode` function

```
/* D(G) : a DAG with the ground G; C : active constraints; D: variable domains */
algorithm FBPD(in : D(G), C; in/out : D)
01:   Reset the node ranges of D(G) to the ranges in either D, C, or [−∞, +∞];
02:   L_f := ∅; L_b := ∅; V_oc := (0, ..., 0); V_ch := (0, ..., 0);
03:   V_lvl := (0, ..., 0); /* this can be made optional together with line 06 */
04:   for each node C representing an active constraint in C do
05:       NodeOccurrences(C, V_oc);
06:       NodeLevel(C, V_lvl); /* this can be made optional */
07:       ForwardEvaluation(C, V_ch, L_b);
08:   end-for
09:   while L_b ≠ ∅ ∨ L_f ≠ ∅ do
10:       N := getNextNode(L_b, L_f);
11:       if I(N) was taken from L_b then
12:           for each child C of N do
13:               BP(N, C); /* see the description of (12) */
14:               if I(C) = ∅ then return infeasible;
15:               if I(C) changed enough for forward evaluation then
16:                   for each P ∈ parents(C) \ {N, G} do
17:                       if V_oc[P] > 0 then put P into L_f;
18:               end-if
19:               if I(C) changed enough for backward propagation then
20:                   Put C into L_b;
21:           end-for
22:       else /* N was taken from L_f */
23:           FE(N, [f]); /* f is the operator at N, see the description of (10) */
24:           if I(N) = ∅ then return infeasible;
25:           if I(N) changed enough for forward evaluation then
26:               for each P ∈ parents(N) \ {G} do
27:                   if V_oc[P] > 0 then put P into L_f;
28:           end-if
29:           if I(N) changed enough for backward propagation then
30:               Put N into L_b;
31:       end-if
32:   end-while
33:   Update D with the ranges of the nodes representing the variables;
end
```

Fig. 4. The Partial Forward-Backward Propagation on a DAG (FBPD) algorithm

```
procedure ForwardEvaluation(in : N; in/out : V_ch, L_b)
    if N is a leaf or V_ch[N] = 1 then return;
    for each child C of node N do ForwardEvaluation(C, V_ch, L_b);
    if N = G then return;
    FE(N, [f]); /* f is the operator at N, similar to line 23 in Figure 4 */
    V_ch[N] := 1; /* the range of this node is cached */
    if I(N) = ∅ then return infeasible;
    if I(N) changed enough for backward propagation then put C into L_b;
end
```

Fig. 5. This is a procedure to do a recursive forward evaluation

```
procedure NodeLevel(in : N; out : Vlvl)
    for each child C of node N do
        Vlvl[C] := max{Vlvl[C], Vlvl[N] + 1};
        NodeLevel(C, Vlvl);
    end-for
end
```

Fig. 6. This is a procedure assigning a node level to each node in a DAG.

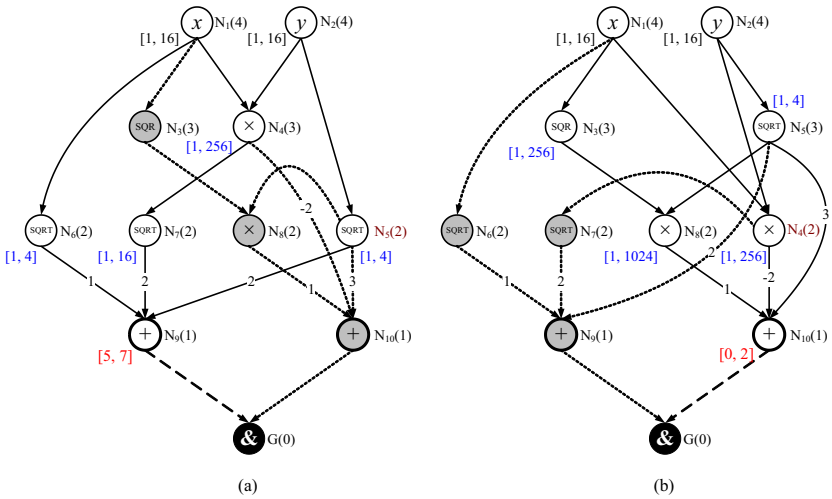


Fig. 7. The node levels are updated at each call to the FBPD algorithm

at line 10 in Figure 4 chooses one of the two nodes at the beginning of $\mathcal{L}_b$ and $\mathcal{L}_f$. The strategy that we use in our implementation is "backward propagation first", that is, taking the node at the beginning of $\mathcal{L}_b$ whenever it is available.

The call to the NodeLevel procedure at line 06 in Figure 4 can be made optional as follows. The first option allows invoking NodeLevel only at the first call to FBPD. The node levels of the initial DAG still meet the requirements on the ordering of the waiting lists. The numbers in brackets next to the node names in Figure 3 are the node levels computed for the initial DAG. Figure 7 illustrates the second option that allows invoking NodeLevel at line 06 in Figure 4 every time FBPD is invoked.

When Are the Changes of Node Ranges Enough? For simplicity, in Figure 4 (lines 15, 19, 25, 29) we only briefly present the procedures to check whether the node ranges have been changed enough for further processing. Hereafter, we will detail them. Let denote by **M** the node **C** at line 13 or the node **N** at line 23. In Figure 4, the forward evaluation at line 23 and the backward propagation at line 13 are of the form

$$\mathbb{I}(\mathbf{M}) := \mathbb{I}(\mathbf{M}) \cap \mathbf{y} \tag{13}$$

where **y** is the interval computed by the forward evaluation or backward propagation before intersecting with $\mathbb{I}(\mathbf{M})$.

Let W_{old} and W_{new} be the widths of $\mathbb{I}(\mathbf{M})$ and $\mathbb{I}(\mathbf{M}) \cap \mathbf{y}$, respectively. In practice, the change of $\mathbb{I}(\mathbf{M})$ after performing (13) is considered enough for doing the forward evaluation at its parents if the conditions $W_{new} < r_f W_{old}$ and $W_{new} + d_f \leq W_{old}$ hold, where r_f is a real number in $(0, 1)$ and d_f is a small positive real number. The numbers r_f and d_f can be predefined or dynamically computed. Similarly, the change of $\mathbb{I}(\mathbf{M})$ after performing (13) is considered enough for doing the backward propagation at $\mathbf{M}$ if the conditions $W_{new} < r_b W_{old}$ and $W_{new} + d_b \leq W_{old}$ hold, where r_b is a real number in $(0, 1)$ and d_b is a small positive real number. Moreover, if $\mathbf{y}$ is computed by the forward evaluation (at line 23), the additional condition $\mathbf{y} \not\subseteq \mathbb{I}(\mathbf{M})$ must also hold.

Proposition 1. *We define a function $F : \mathbb{I}^n \times 2^{\mathbb{R}^n} \rightarrow \mathbb{I}^n$ to represent the* FBPD *algorithm. This function takes as input the variable domains (in the form of an interval box $\mathbf{B}$) and the exact solution set, S, of the input problem. The function F returns an interval box, denoted by $F(\mathbf{B}, S)$, that represents the variable domains of the output of the* FBPD *algorithm. If the input problem contains only the elementary operations f defined in Definition 3, then the* FBPD *algorithm terminates at a finite number of iterations and the following properties hold:*

$$(i)\ \ F(\mathbf{B}, S) \subseteq \mathbf{B} \qquad \textit{(Contractiveness)}$$
$$(ii)\ F(\mathbf{B}, S) \supseteq \mathbf{B} \cap S \ \textit{(Completeness)}$$

Proof. The proof is easy, and is therefore omitted due to lack of space.

4.2 Combining Propagation and Search Using a DAG

The most common framework for solving NCSPs is the branch-and-prune framework. The most common exhaustive search is the bisection. Bisection search is suitable for problems with isolated solutions. However, it is often inefficient for problems with a continuum of solutions. Therefore, for the problems with a continuum of solutions we need more advanced search techniques like `UCA5`, `UCA6` and `UCA6+` (see [6, 7]). They all can be seen as instances of the generic branch-and-prune search described in Figure 8. In general, the search scheme produces two lists. The first list, $\mathcal{L}_\forall$, consists of feasible sub-domains. The second list, $\mathcal{L}_\varepsilon$, consists of tuples of tiny sub-domains, that are smaller than the required resolution ε, and the sets of constraints, that are still active in the sub-domains.

5 Experiments

We have carried out experiments on `FBPD` and two other recent interval propagation techniques. The first one is Box Consistency implemented in a commercial product named ILOG Solver (v6.0, 11/2003), hereafter denoted by `BOX`. The second one is called `HC4` (Revised Hull Consistency) from [4]. The experiments are carried out on 33 problems which are unbiasedly selected and divided into 5 test cases. The test case T_1 consists of 8 problems with isolated solutions that are solvable by all three propagators. The test case T_2 consists of 4 problems with isolated solutions that are solvable by only two propagators. The test case

```
algorithm BnPSearch(in : V, C, D; out : L∀, Lε)
    Construct a DAG, D(G), for the initial problem (V, C, D);
    FPBD(D(G), C, D); /* Prune the domains in D */
    if infeasible is detected then return infeasible;
    if domains in D are small enough then Lε := Lε ∪ {(C, D)}; return;
    L := {(C, D)};
    while L ≠ ∅ do
        Get a couple (C0, D0) from L;
        Split the problem (V, C0, D0) into subproblems {(V, Ci, Di)}_{i=1}^k; where Ci ⊆ C0
        for i := 1 to k do
            FPBD(D(G), Ci, Di); /* Prune the domains in Di */
            if infeasible is detected then continue for;
            if Ci = ∅ then L∀ := L∀ ∪ {Di}; continue for;
            if domains in Di are small enough then
                Lε := Lε ∪ {(Ci, Di)}
            else
                L := L ∪ {(Ci, Di)};
            end-if
        end-for
    end-while
end
```

Fig. 8. A generic branch-and-prune search using FPBD for pruning.

T_3 consists of 8 problems with isolated solutions that cause at least two of three techniques to stop due to timeout or due to running more than 10^6 iterations. The test case T_4 consists of 7 small problems with continuum of solutions that are solvable at resolution 10^{-2}. The test case T_5 consists of 6 hard problems with continuum of solutions that are solvable at resolution 10^{-1}. The timeout value is 2 hours for all the test cases, it will be used as the running time for the techniques which are timeout in the next result analysis (i.e. we are in favor of slow techniques). For the first three test cases, the resolution is 10^{-4} and the search to be used is bisection. For the last two test cases, the search to be used is a simple search technique, called UCA6, for inequalities (see [6, 7]). The comparison of the interval propagation techniques is based on the measures of

1. *The running time:* The relative ratio of the running time of each propagator to that of FBPD is called the *relative time ratio*.
2. *The number of boxes:* The relative ratio of that number of boxes in the output of each propagator to that of FBPD is called the *relative cluster ratio*.
3. *The number of iterations:* the number of iterations in search needed to solve the problems. The relative ratio of the number of iterations used by each propagator to that of FBPD is called the *relative iteration ratio*.
4. *The volume of boxes (only for T_1, T_2, T_3):* We consider the reduction per dimension when replacing the set of output boxes by a volume-equivalent hypercube. The relative ratio of the reduction gained by each propagator to that of FBPD is called the *relative reduction ratio*.
5. *The volume of inner boxes (only for T_4, T_5):* The ratio of the volume of inner boxes to the volume of all output boxes is called the *inner volume ratio*.

Table 1. (*a*) The average of the relative time ratios is taken over all the problems in the test cases T_1, T_2, T_3; the averages of the other relative ratios are taken over the problems in the test case T_1, i.e. over the problems which are solvable by all the techniques. (*b*) The averages of the relative ratios are taken over all the problems in the test cases T_4, T_5. In general, the lower the relative ratio, the better the performance/quality; and the higher the inner volume ratio, the better the quality.

Propagator	(*a*) *Isolated Solutions*				(*b*) *Continuum of Solutions*			
	Relative time ratio	Relative reduction ratio	Relative cluster ratio	Relative iteration ratio	Relative time ratio	Inner volume ratio	Relative cluster ratio	Relative iteration ratio
FBPD	**1.000**	1.000	1.000	1.000	**1.000**	0.922	1.000	1.000
BOX	20.863	**0.625**	**0.342**	**0.731**	20.919	**0.944**	**0.873**	**0.854**
HC4	203.285	0.906	1.266	0.988	403.915	0.941	0.896	0.879

Table 2. This table contains the averages of the relative time ratios taken over the problems in each test case.

Propagator	(*a*) *Isolated Solutions*			(*b*) *Continuum of Solutions*	
	Test case T_1	Test case T_2	Test case T_3	Test case T_4	Test case T_5
FBPD	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
BOX	24.21	28.98	13.45	11.55	31.85
HC4	94.42	691.24	68.17	191.86	651.31

Table 3. This table contains the overrun ratios for the test case T_1. An overrun ratio greater than 1 would suffice for applications.

Problem →	BIF-3	REI-3	WIN-3	ECO-5	ECO-6	NEU-6	ECO-7	ECO-8	**Average**
FBPD	1.626	1.360	2.075	1.711	1.676	3.198	1.513	1.455	**1.880**
BOX	2.957	1.974	3.080	1.579	1.660	6.748	1.521	1.485	**2.625**
HC4	2.229	1.914	1.492	1.647	1.679	4.949	1.488	1.449	**2.106**

The overviews of results in our experiments are given in Tables 1 and 2. In Table 3, we give the *overrun ratio* of each propagator for the test case T_1. The overrun ratio is defined by $\varepsilon / \sqrt[d]{V/N}$; where ε is the required resolution, d is the dimension of the problem, V is the total volume of the output boxes, N is the number of output boxes. Clearly, FBPD outperforms both BOX and HC4 by an order of magnitude or more in speed, while being roughly the same in the quality measures in case where the solution set to be enclosed by boxes of macroscopic size (i.e. for continuum of solutions). For isolated solutions, very narrow boxes are produced by any technique in comparison to the required resolution. However, the new technique is 1.1–2.0 times less tight than the other techniques in the measure on reduction per dimension (which hardly matters in applications). In comparison with HC4, a constraint propagation technique that is similar to FBPD but works on the tree representation instead of DAGs, FBPD is clearly more suitable for applications.

6 Conclusion

We propose a constraint propagation technique, FBPD, which makes the fundamental framework for constraint propagation on DAGs [1] efficient and practical, and a method to coordinate constraint propagation and exhaustive search using a single DAG for each problem. The experiments carried out on various problems

show that the new approach outperforms previously available propagation techniques by an order of magnitude or more in speed, while being roughly the same quality w.r.t. enclosure properties. In other views, `FBPD` can be seen as a special instance of a generic combination scheme, `CIRD`, proposed by Vu *et al.* [13]. Moreover, our experiments show that the strengths of `FBPD` and `CIRD1` (an instance of the `CIRD` scheme) are complementary, therefore, unifying their strengths is a straightforward direction in the near future.

Acknowledgements. This research was funded by the European Commission and the Swiss Federal Education and Science Office through the COCONUT project (IST-2000-26063). We would like to thank ILOG for the licenses of ILOG Solver used in the COCONUT project, and thank IRIN team at University of Nantes for the `HC4` code. We also thank Prof. Arnold Neumaier for fruitful discussions and very valuable input.

References

1. Schichl, H., Neumaier, A.: Interval Analysis on Directed Acyclic Graphs for Global Optimization (2004) preprint - University of Vienna, Autria.
2. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. First edn. Springer (2001)
3. Granvilliers, L., Goualard, F., Benhamou, F.: Box Consistency through Weak Box Consistency. In: Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'99). (1999) 373–380
4. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising Hull and Box Consistency. In: Proceedings of the International Conference on Logic Programming (ICLP'99), Las Cruces, USA (1999) 230–244
5. Benhamou, F., Goualard, F.: Universally Quantified Interval Constraints. In: Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000). (2000) 67–82
6. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Search Techniques for Non-linear CSPs with Inequalities. In: Proceedings of the 14th Canadian Conference on Artificial Intelligence. (2001)
7. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In: Global Optimization and Constraint Satisfaction. Volume LNCS 2861., Springer-Verlag (2003) 194–210
8. Kearfott, B.R.: Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin **2(1)** (1996) 95–112
9. Burkill, J.C.: Functions of Intervals. Proceedings of the London Mathematical Society **22** (1924) 375–446
10. Moore, R.E.: Interval Arithmetic and Automatic Error Analysis in Digital Computing. PhD thesis, Stanford University, USA (1962)
11. Hickey, T.J., Ju, Q., Van Emden, M.H.: Interval Arithmetic: from Principles to Implementation. Journal of the ACM (JACM) **48(5)** (2001) 1038–1068
12. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge Univ. Press, Cambridge (1990)
13. Vu, X.H., Sam-Haroud, D., Faltings, B.: A Generic Scheme for Combining Multiple Inclusion Representations in Numerical Constraint Propagation. Technical Report IC/2004/39, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland (2004)

Combining Multiple Inclusion Representations in Numerical Constraint Propagation

Xuan-Ha Vu, Djamila Sam-Haroud, and Boi V. Faltings

Artificial Intelligence Laboratory,
Swiss Federal Institute of Technology in Lausanne (EPFL),
CH-1015, Lausanne, Switzerland
{xuan-ha.vu, jamila.sam, boi.faltings}@epfl.ch
http://liawww.epfl.ch

Abstract. This paper proposes a novel generic scheme enabling the combination of multiple inclusion representations to propagate numerical constraints. The scheme allows bringing into the constraint propagation framework the strength of inclusion techniques coming from different areas such as interval arithmetic, affine arithmetic or mathematical programming. The scheme is based on the DAG representation of the constraint system. This enables devising fine-grained combination strategies involving any factorable constraint system. The paper presents several possible combination strategies for creating practical instances of the generic scheme. The experiments reported on a particular instance using interval propagation, interval arithmetic, affine arithmetic and linear programming illustrate the flexibility and efficiency of the approach.

1 Introduction

Many real world applications involve the solving of problems modeled as *numerical constraints* on variables with *continuous* domains. In practice, numerical constraints can be equalities or inequalities of arbitrary type, usually expressed in *factorable* form, that is, they can be represented by elementary functions such as $+$, $-$, $\times$, $\div$, log, exp, sin, cos, ... Recently, many solution techniques have been proposed in *constraint programming* to solve such constraint systems. Some of them are based on *interval (constraint) propagation* and *interval arithmetic* (e.g. the works in [1, 2] and references therein), while some of them are based on *linear relaxation* and *linear programming* [3, 4]. There have also been mathematical solving techniques [5, 6] that use *G interval* or *affine arithmetic* to solve equation systems. Most of the solution techniques are interleaved with a *bisection* search to solve the problems exhaustively. Lately, there have been some advanced search techniques [7, 8] that improve the search performance for problems with non-isolated solutions (e.g., inequalities) while maintaining the same performance for problems with isolated solutions (e.g., equalities). In general, different techniques have different strengths that are complementary. Therefore, combining the strength of different solution techniques is the subject of many intensive research efforts (see [2] and references therein).

Our contributions will be described in Section 3 and Section 4. In Section 3, we propose a novel generic scheme which allows devising new combination strategies for numerical constraint propagation in a flexible way. The scheme enables the propagation to be performed using different inclusion representations on a *directed acyclic graph* (DAG) that represents the problem. The goal is to provide a combination scheme that is efficient and flexible but still general enough to bring the strength of different solution techniques coming from different areas (e.g., constraint programming and mathematical programming) into the framework of constraint propagation. In order to illustrate the flexibility and efficiency of the proposed scheme, in Section 4 we propose improvements on affine arithmetic and then devise from the scheme several new combination strategies which are based on emerging techniques, namely interval constraint propagation, interval arithmetic, affine arithmetic, and linear programming. In Section 5, our experiments show that the devised technique is superior than the recent interval propagation methods in performance and quality. It even outperforms some very recent techniques in mathematical programming and constraint programming which are specially designed to solve certain constraint systems. Finally, the conclusion and future directions are given in Section 6.

2 Background

2.1 Interval Arithmetic and Affine Arithmetic

When using an interval $[a, b] \subseteq \mathbb{R}$ to represent a quantity x, we mean that

$$a \leq x \leq b \tag{1}$$

Interval arithmetic is an arithmetic defined on the set of intervals, rather than the set of real numbers. Modern interval arithmetic was originated independently in late 1950s by several researchers; including M. Warmus (1956), T. Sunaga (1958) and R. E. Moore (1959). We assume that readers are familiar with interval arithmetic. Otherwise, we would recommend [9, 2] and references therein for more details on interval arithmetic and basic interval methods.

Affine arithmetic [10] is an extension of interval arithmetic which keeps track of correlations between computed and input quantities. A real quantity x is represented by an *affine form* which is a first-degree polynomial of the form

$$x = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n \tag{2}$$

where $x_0, \ldots, x_n$ are real coefficients and $\varepsilon_1, \ldots, \varepsilon_n$ are *noise variables* (originally called *noise symbols*) taking values in $[-1, 1]$.

Similarly to interval arithmetic, affine arithmetic also allows to use rounded floating-point arithmetic to construct *rigorous enclosures* for the ranges of operations and functions [11]. In affine arithmetic, affine operations such as $\alpha x+\beta y+\gamma$ are obtained exactly, except the rounding errors, by the following formula:

$$\alpha x + \beta y + \gamma = (\alpha x_0 + \beta y_0 + \gamma) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\varepsilon_i \tag{3}$$

However, non-affine operations can only be computed by approximations. In general, the exact result of a non-affine operation has form $f^*(\varepsilon_1, \ldots, \varepsilon_n)$, where

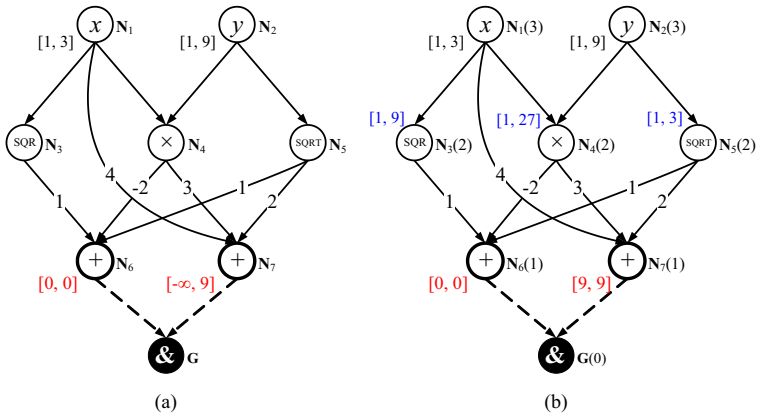


Fig. 1. The DAG representation (a) before and (b) after interval evaluations

f^* is a non-linear function. In practice, this result is then approximated by an affine function $f^a(\varepsilon_1, \ldots, \varepsilon_n) = z_0 + z_1\varepsilon_1 + \ldots + z_n\varepsilon_n$. A new term $z_k\varepsilon_k$ is used to represent the difference between f^* and f^a, hence, the result becomes affine:

$$z = z_0 + z_1\varepsilon_1 + \ldots + z_n\varepsilon_n + z_k\varepsilon_k \tag{4}$$

where the maximum absolute error $z_k \geq \sup\{|f^*(\varepsilon_1, \ldots, \varepsilon_n) - f^a(\varepsilon_1, \ldots, \varepsilon_n)| : \forall(\varepsilon_1, \ldots, \varepsilon_n) \in [-1, 1]^k\}$. An important goal is to keep this maximum absolute error as small as possible (see the *Chebyshev approximation theory*).

Ranges obtained with affine arithmetic may be more accurate than those obtained with interval arithmetic. However, the operations of affine arithmetic are often more expensive than those of interval arithmetic. Some comparisons on interval and affine arithmetic methods can be found in [11–13].

2.2 Directed Acyclic Graph

We assume that readers are already familiar with fundamental concepts in graph theory like *directed multigraph with ordered edges* and *directed acyclic graph/multigraph*. Otherwise, readers are referred to [14].

Theorem 1. *For every directed acyclic multigraph (V, E, f) there exists a total order $\preceq$ on vertices V such that $\forall v \in V$: if u is an ancestor of v, then $v \preceq u$.*

Following the approach in [14], we use a directed acyclic multigraph, whose edges are totally ordered and whose vertices are ordered by an order in Theorem 1, to represent a constraint system, we therefore call it a *directed acyclic graph* (DAG). In a DAG, every node represents a variable or an elementary operation such as $+$, $\times$, $\div$, log, exp, ... and every edge represents the computational flow. The ordering of edges is needed for non-commutative operations like the division, and not really necessary for commutative operations. For convenience, a virtual ground node is added to a DAG to be the parent of all the nodes representing the constraints. For example, the DAG representation of the constraint system $\{x^2 - 2xy + \sqrt{y} = 0,\ 4x + 3xy + 2\sqrt{y} \leq 9 \mid x \in [1, 3], y \in [1, 9]\}$ is given in Figure 1, where $\{\mathbf{N}_1, \mathbf{N}_2, \mathbf{N}_3, \mathbf{N}_4, \mathbf{N}_5, \mathbf{N}_6, \mathbf{N}_7\}$ is an ordering of the nodes.

3 Combination Scheme for Constraint Propagation

3.1 Generalization of Inclusion Concepts

We generalize the concepts related to *inclusion function* as follows.

Definition 1 (Inclusion Representation). *Given a set $\mathcal{A}$. A couple $\mathcal{I} = (\mathcal{R}, \mu)$, where $\mathcal{R}$ is a set of* representing objects *and μ is a function from $\mathcal{R}$ to $2^{\mathcal{A}}$, is called an* inclusion representation *of $\mathcal{A}$ if there exists a surjective function $\rho : 2^{\mathcal{A}} \to \mathcal{R}$ such that $\forall S \subseteq \mathcal{A} : S \subseteq \mu(\rho(S))$. In this case, ρ is called the* representing function *of $\mathcal{I}$ and μ is called the* evaluating function *of $\mathcal{I}$.*

Let $\mathcal{I} = (\mathcal{R}, \mu)$ be an inclusion representation of $\mathbb{R}$. We call $\mathcal{I}$ a *real inclusion representation* of $\mathbb{R}$ if each representing object $T \in \mathcal{R}$ is a tuple consisting of real constants, and the evaluating function μ can be defined by

$$\mu(T) \equiv \{f_T(V_T) \mid V_T \in D_T\} \tag{5}$$

where f_T is a real-valued function (with T as a tuple of parameters) and V_T is a finite sequence of variables taking values in real domains D_T. The representation (5) is called a *real representation* of μ.

It is easy to see that the interval form (1) is a real inclusion representation of $\mathbb{R}$, where each representing object $T \in \mathcal{R}$ is a couple of reals (a, b), $V_T = (x)$, f_T is an identity function, and μ is defined by

$$\mu(T) \equiv \{x \mid x \in [a, b]\} \tag{6}$$

The affine form (2) is also a real inclusion representation of $\mathbb{R}$, where each representing object is a tuple $T = (x_0, \ldots, x_n, 1, \ldots, n)$,[1] and $V_T = (\varepsilon_1, \ldots, \varepsilon_n)$ are the variables of the linear function $f_T(\varepsilon_1, \ldots, \varepsilon_n) = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n$. Hence, the real representation of the evaluating function is defined by

$$\mu(T) \equiv \{x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n \mid (\varepsilon_1, \ldots, \varepsilon_n) \in [-1, 1]^n\} \tag{7}$$

Definition 2 (Inclusion Function). *Given two sets X, Y and a function $f : X \to Y$. Let $\mathcal{I}_X = (\mathcal{R}_X, \mu_X)$ and $\mathcal{I}_Y = (\mathcal{R}_Y, \mu_Y)$ be two inclusion representations of X and Y, respectively. A function $F : \mathcal{R}_X \to \mathcal{R}_Y$ is called an* inclusion function *of f, if $\forall S \subseteq X, \forall T \in \mathcal{R}_X : S \subseteq \mu_X(T) \Rightarrow \{f(x) \mid x \in S\} \subseteq \mu_Y(F(T))$.*

Definition 3 (Inclusion Converter). *Let $\mathcal{I}_1 = (\mathcal{R}_1, \mu_1)$ and $\mathcal{I}_2 = (\mathcal{R}_2, \mu_2)$ be two inclusion representations of the same set. A function $c : \mathcal{R}_1 \to \mathcal{R}_2$ is called an* inclusion converter from $\mathcal{I}_1$ to $\mathcal{I}_2$ *if $\forall S \in \mathcal{R}_1 : \mu_1(S) \subseteq \mu_2(c(S))$.*

Theorem 2. *Let $\mathcal{I}_X = (\mathcal{R}_X, \mu_X)$, $\mathcal{I}_Y = (\mathcal{R}_Y, \mu_Y)$ and $\mathcal{I}_Z = (\mathcal{R}_Z, \mu_Z)$ be inclusion representations of three sets X, Y and Z, respectively. If $F : \mathcal{R}_X \to \mathcal{R}_Y$ and $G : \mathcal{R}_Y \to \mathcal{R}_Z$ are inclusion functions of two functions $f : X \to Y$ and $g : Y \to Z$ respectively, then the composite function $G \circ F$ is an inclusion function of the composite function $g \circ f$.*

Corollary 1. *Let $\mathcal{I} = (\mathcal{R}, \mu)$ be an inclusion representation of $\mathbb{R}$. If all elementary operations defined on $\mathcal{R}$ are inclusion functions of their counterparts on $\mathbb{R}$, then all factorable functions built on $\mathcal{R}$ are also inclusion functions of their counterparts on $\mathbb{R}$.*

[1] In implementation, only non-zero coefficients and their indices should be stored.

The proof of Theorem 2 follows directly from Definition 1 and Definition 2. Corollary 1 is a straightforward consequence of Theorem 2. The elementary operations in interval arithmetic and affine arithmetic are inclusion functions of their real-valued counterparts, therefore, as a result of Corollary 1, all the factorable operations/functions defined in interval arithmetic (or affine arithmetic) are also inclusion functions of their real-valued counterparts.

3.2 A General Combination Scheme

In this section, we describe a general combination scheme that combines the strength of different real inclusion representations for constraint propagation. In this scheme, the input constraint system is represented by a DAG as described in Section 2.2. The data stored at each node, $\mathbf{N}$, of the DAG consist of a representing object for each real inclusion representation $\mathcal{I} = (\mathcal{R}, \mu)$ of $\mathbb{R}$ and a *constraint range* $\tau(\mathbf{N}) \subseteq \mathbb{R}$, i.e. the node range of a constraint, is an interval. We denote by $\mathcal{R}(\mathbf{N})$ the representing object (in $\mathcal{I}$) stored at $\mathbf{N}$.

Definition 4 (Inclusion Constraint System, ICS). *Let $(\mathcal{R}, \mu)$ be a real inclusion representation of $\mathbb{R}$ defined by* (5)*, $\mathbf{N}$ a node of a DAG representing a constraint system. The* inclusion constraint system *induced by a representing object $T \equiv \mathcal{R}(\mathbf{N})$ and a* constraint domain *$D \subseteq \mathbb{R}$ is defined by*

$$\texttt{ICS}(T, D) \equiv \begin{cases} \{\vartheta_{\mathbf{N}} \in D_T, \vartheta_{\mathbf{N}} \in D\} & \textit{(where } V_T \equiv \{\vartheta_{\mathbf{N}}\}\textit{) if } f_T \textit{ is identity,} \\ \{f_T(V_T) = \vartheta_{\mathbf{N}}, V_T \in D_T, \vartheta_{\mathbf{N}} \in D\} & \textit{otherwise;} \end{cases}$$

where $\vartheta_{\mathbf{N}}$ (i.e. the representing variable of $\mathbf{N}$) and the variables in V_T are the variables of the constraint system.

Definition 5 (NEV, PCS). *Let $\mathbf{N}$ be a node of a DAG representing a constraint system, $\{\mathbf{C}_i\}_{i=1}^k$ the children of $\mathbf{N}$, $f : \mathbb{R}^k \to \mathbb{R}$ the elementary operation represented by $\mathbf{N}$, $\mathcal{S}$ a finite set of real inclusion representations. The following constraint system is called the* pruning constraint system *in $\mathcal{S}$ at $\mathbf{N}$:* $\texttt{PCS}(\mathbf{N}, \mathcal{S}) \equiv$

$$\begin{cases} \{\bigwedge_{i=1}^k \texttt{ICS}(\mathcal{R}(\mathbf{C}_i), \tau(\mathbf{C}_i))\} & \textit{if } \mathbf{N} \textit{ is ground,} \\ \left\{ \begin{array}{l} f(\vartheta_{\mathbf{C}_1}, \ldots, \vartheta_{\mathbf{C}_k}) = \vartheta_{\mathbf{N}} \wedge \\ \bigwedge_{(\mathcal{R},\mu)\in\mathcal{S}} (\texttt{ICS}(\mathcal{R}(\mathbf{N}), \tau(\mathbf{N})) \wedge \bigwedge_{i=1}^k \texttt{ICS}(\mathcal{R}(\mathbf{C}_i), \tau(\mathbf{C}_i))) \end{array} \right\} & \textit{otherwise.} \end{cases}$$

For each $\mathcal{I} = (\mathcal{R}, \mu) \in \mathcal{S}$, let $f_{\mathcal{I}} : \mathcal{R}^k \to \mathcal{R}$ be an inclusion function of f. The following assignment is called the node evaluation *in $\mathcal{I}$ at $\mathbf{N}$ (if $\mathbf{N} \neq$ ground):*

$$\texttt{NEV}(\mathbf{N}, \mathcal{I}) \equiv \left\{ \begin{array}{l} \mathcal{R}(\mathbf{N}) := \mathcal{R}(\mathbf{N}) \cap \tau(\mathbf{N}) \cap f_{\mathcal{I}}(\mathcal{R}(\mathbf{C}_1), \ldots, \mathcal{R}(\mathbf{C}_k)); \\ \tau(\mathbf{N}) := \tau(\mathbf{N}) \cap \mu(\mathcal{R}(\mathbf{N})); \end{array} \right\}$$

A technique is called a *pruning technique* for a real constraint system if it is capable of reducing some domains of the variables in that system. Let $\mathcal{G}$ be a DAG representing the input constraint system. The following algorithm scheme, called CIRD, uses two waiting lists. The first waiting list stores the nodes waiting for evaluation, denoted by $\mathcal{L}_e$. The second waiting list stores the nodes waiting for node pruning, denoted by $\mathcal{L}_p$. Note that each node can appear once at a time

in a waiting list. The set of real inclusion representations for use in the scheme is denoted by $\mathcal{E}$. Each real inclusion representation in $\mathcal{E}$ provides the elementary operations that are inclusion functions of their real-valued counterparts. The following gives the main steps of the CIRD scheme. It is easy to see that the contractor following the CIRD scheme terminates at a finite number of steps, moreover, it is *contractive* and *complete* as expected.

A Scheme for Combining Inclusion Representations on DAG (CIRD)

1. **Initialization Phase.**
   (a) **Initial Node Evaluation.** Select an algorithm for visiting DAGs in an ordering described in Theorem 1. When visiting a node $\mathbf{N} \in \mathcal{G}$, perform the node evaluation $\texttt{NEV}(\mathbf{N}, \mathcal{I})$ for each $\mathcal{I} \in \mathcal{E}$.
   (b) **Initialize Waiting Lists.** Set $\mathcal{L}_e := \emptyset$, $\mathcal{L}_p :=$ {the list of all nodes representing the active constraints together with all the real inclusion representations of $\mathcal{E}$}.
2. **Propagation Phase.** Repeat this step until both $\mathcal{L}_e$ and $\mathcal{L}_p$ become empty.
   (a) **Get the Next Node.** Select a strategy for getting a node $\mathbf{N}$ (and the set $\mathcal{S}$ of real inclusion representations associated with $\mathbf{N}$ in the corresponding list) from the two waiting lists $\mathcal{L}_e$ and $\mathcal{L}_p$.
   (b) **Node Evaluation.** *Do this step only if* $\mathbf{N}$ *was taken from* $\mathcal{L}_e$ *at Step 2a.* For each $\mathcal{I} = (\mathcal{R}, \mu) \in \mathcal{S}$ do the following steps:[2]
      i. Perform the node evaluation $\texttt{NEV}(\mathbf{N}, \mathcal{I})$.
      ii. If the changes of $\mathcal{R}(\mathbf{N})$ and $\tau(\mathbf{N})$ at Step 2(b)i are considered enough to re-evaluate the parents of $\mathbf{N}$, then put each node in $parents(\mathbf{N})$ (associated with $\mathcal{I}$) into $\mathcal{L}_e$, if $\mathbf{N}$ is not the ground node, or into $\mathcal{L}_p$ otherwise.
      iii. If the changes of $\mathcal{R}(\mathbf{N})$ and $\tau(\mathbf{N})$ at Step 2(b)i are considered enough to do a node pruning at $\mathbf{N}$ again, then put $(\mathbf{N}, \mathcal{I})$ into $\mathcal{L}_p$.
   (c) **Node Pruning.** *Do this step only if* $\mathbf{N}$ *was taken from* $\mathcal{L}_p$ *at Step 2a.*
      i. Select a subset $\mathcal{T} \subseteq \mathcal{S}$ such that for each $\mathcal{I} \in \mathcal{T}$ there are efficient pruning techniques for the constraint system $\texttt{PCS}(\mathbf{N}, \mathcal{I})$.
      ii. Partition $\mathcal{T}$ into subsets such that for each subset $\mathcal{U}$ of the partition there is a pruning technique that may efficiently reduce the domains of the variables of the system (or a subsystem of) $\texttt{PCS}(\mathbf{N}, \mathcal{U})$. After that, apply the associated pruning technique to each system (or a subsystem of) $\texttt{PCS}(\mathbf{N}, \mathcal{U})$ in a certain order.
      iii. Let $\mathcal{K}$ be the set of all the nodes whose evaluating functions in form (5) contain some variables whose domains were reduced at Step 2(c)ii. Select a subset $\mathcal{H}$ of $\mathcal{K}$, *for example*, such that each node $\mathbf{M}$ in $\mathcal{H}$ is a descendant of $\mathbf{N}$. For each real inclusion representation $\mathcal{I} = (\mathcal{R}, \mu) \in \mathcal{H}$ such that the representation of $\mu(\mathcal{R}(\mathbf{M}))$ in form (5) contains some variables whose domains were reduced at Step 2(c)ii, update $\mathcal{R}(\mathbf{M})$ using those newly reduced domains, then update $\tau(\mathbf{M}) := \tau(\mathbf{M}) \cap \mu(\mathcal{R}(\mathbf{M}))$.
         A. If the changes of $\mathcal{R}(\mathbf{M})$ and $\tau(\mathbf{M})$ are considered enough to re-evaluate $\mathbf{M}$'s parents, put each node in $parents(\mathbf{M})$ associated with $\mathcal{I}$ into $\mathcal{L}_e$.
         B. If the changes of $\mathcal{R}(\mathbf{M})$ and $\tau(\mathbf{M})$ are considered enough to do a node pruning at $\mathbf{M}$, put $(\mathbf{M}, \mathcal{I})$ into $\mathcal{L}_p$.

[2] Combining several inclusion representations for better evaluation by using inclusion converters is also an option to try.

4 Specific Combination Strategies as Instances of CIRD

In the rest of the paper, we denote by $\lfloor E \rfloor$ (resp. $\lceil E \rceil$) some lower approximation (reps. some upper approximation) in $\mathbb{F}$ of the expression E such that $\lfloor E \rfloor \leq E$ (resp. $E \leq \lceil E \rceil$). We use the notion $E = \langle E \rangle \pm e$ to mean that $\langle E \rangle$ is an approximation in $\mathbb{F}$ of E, and the corresponding bound on the absolute rounding error is e, that is $\langle E \rangle - e \leq E \leq \langle E \rangle + e$. Readers are referred to [11] for some rounding techniques in floating-point arithmetic on simple elementary operations.

4.1 Modifications To Affine Arithmetic

Revised Affine Form. One of the limits of the standard affine arithmetic is that the number of noise symbols grows gradually during the computation and the computation cost heavily depends on this number. Inspired by the ideas in [12, 5, 6], we use a revised affine form similar to (4) but the new term $z_k\varepsilon_k$ is replaced by a non-negative accumulative error $[-e_z, e_z]$ which represents the maximum absolute error z_k of non-affine operations. In other words, the *revised affine form* of a real-valued quantity $\hat{x}$ is defined by

$$\hat{x} \equiv x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n + e_x[-1, 1] \tag{8}$$

which consists of two separated parts: the standard affine part of length n, and the interval part. Where the magnitude of the accumulative error, $e_x \geq 0$, is represented by the interval part. That is, for each value x of the quantity $\hat{x}$ (say $x \in \hat{x}$), there exist $\varepsilon_x \in [-1, 1], \varepsilon_i \in [-1, 1]$ $(i = 1, \ldots, n)$ such that $x = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n + e_x\varepsilon_x$. We then say it is of length n. The affine operations are now defined by

$$\hat{z} \equiv \alpha\hat{x} + \beta\hat{y} + \gamma \equiv (\alpha x_0 + \beta y_0 + \gamma) + \sum_{i=1}^{n} (\alpha x_i + \beta y_i)\varepsilon_i + (|\alpha|e_x + |\beta|e_y)[-1, 1] \tag{9}$$

Therefore, during the computation the length of revised affine forms will not exceed the number of noise symbols at the beginning, i.e. the number of variables of the input constraint system. In rigorous computing, e_z will be used to accumulate the rounding errors in floating-point arithmetic, namely (9) can be interpreted as follows

$$z_0 = \langle \alpha x_0 + \beta y_0 + \gamma \rangle \pm e_0,\ z_i = \langle \alpha x_i + \beta y_i \rangle \pm e_i,\ e_z = \lceil |\alpha|e_x + |\beta|e_y + \sum_{i=0}^{n} e_i \rceil \tag{10}$$

Another limit of the standard affine form is that it is not capable of handling half-lines of the form $(-\infty, a]$ and $[a, +\infty)$, while this is important for many computation methods, especially constraint propagation and search techniques. Hence, we propose to associate each quantity $\hat{x}$ with a data field $x_\infty \in \{-1, 0, +1\}$. The revised affine form is then interpreted as follows:[3]

$$\hat{x} \equiv \begin{cases} (-\infty, +\infty) & \text{if } e_x = +\infty, \\ (-\infty, x_0] & \text{if } x_\infty = -1, \\ [x_0, +\infty) & \text{if } x_\infty = +1, \\ x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n + e_x[-1, 1] & \text{otherwise.} \end{cases} \tag{11}$$

[3] For simplicity, we allow zero coefficients in the formulae in the paper, however in implementation one should keep only nonzero coefficients.

Table 1. Examples of functions $f \in \mathcal{C}^1([a,b])$ satisfying the conditions of Theorem 3

$f(x)$	$[a,b]$ is a subset of	$f'(x)$	f'	$g(\alpha)$
$\sqrt{x}$	$[0,+\infty)$	$1/(2\sqrt{x})$	$\downarrow$	$1/(4\alpha) : \alpha > 0$
e^x	$(-\infty,+\infty)$	e^x	$\uparrow$	$\alpha(1-\log\alpha) : \alpha > 0$
$\log x$	$(0,+\infty)$	$1/x$	$\downarrow$	$-(1+\log\alpha) : \alpha > 0$
$x^n : n \geq 2$ is even	$(-\infty,+\infty)$	nx^{n-1}	$\uparrow$	$(1-n)\sqrt[n-1]{(\alpha/n)^n}$
$x^n : n \geq 3$ is odd	$(-\infty,0]$	nx^{n-1}	$\downarrow$	$(n-1)\sqrt[n-1]{(\alpha/n)^n} : \alpha \geq 0$
$x^n : n \geq 3$ is odd	$[0,+\infty)$	nx^{n-1}	$\uparrow$	$(1-n)\sqrt[n-1]{(\alpha/n)^n} : \alpha \geq 0$
$1/x^n : n \geq 2$ is even	$(-\infty,0);(0,+\infty)$	$-n/x^{n+1}$	$\uparrow$	$(n+1)\sqrt[n+1]{(-\alpha/n)^n}$
$1/x^n : n \geq 1$ is odd	$(-\infty,0)$	$-n/x^{n+1}$	$\downarrow$	$-(n+1)\sqrt[n+1]{(-\alpha/n)^n} : \alpha < 0$
$1/x^n : n \geq 1$ is odd	$(0,+\infty)$	$-n/x^{n+1}$	$\uparrow$	$(n+1)\sqrt[n+1]{(-\alpha/n)^n} : \alpha < 0$
$x^r : r \notin [0,1]$	$(0,+\infty)$	rx^{r-1}	$\uparrow$	$(1-r)(\alpha/r)^{(r/(r-1))} : \alpha r > 0$
$x^r : r \in (0,1)$	$(0,+\infty)$	rx^{r-1}	$\downarrow$	$(1-r)(\alpha/r)^{(r/(r-1))} : \alpha > 0$

In an operation, if the domain of a variable is unbounded, i.e. in the first three cases of (11), the other variables are converted into interval forms for that operation performed in interval arithmetic, then the result is converted back to affine form. Therefore, in the rest of paper, we only need to discuss about the last case of (11). The set of all objects in revised affine form is denoted by $\mathbb{A}$.

Unary Operations. We give the following theorem as a basis for finding affine approximations of elementary univariate functions in a rigorous manner.

Theorem 3 (Affine Approximation of Univariate Functions). *Let f be a differentiable function on $[a,b]$, where $a < b$ in $\mathbb{R}$, and $d_\alpha(x) \equiv f(x) - \alpha x$.*

1. *If $\forall x \in [a,b] : \alpha \geq f'(x)$, then $\forall x \in [a,b] : \alpha x + d_\alpha(b) \leq f(x) \leq \alpha x + d_\alpha(a)$.*
2. *If f' is continuous and monotone increasing on $[a,b]$, we have*
   (a) *$\forall \alpha \in [f'(a), f'(b)], \exists c \in [a,b] : f'(c) = \alpha$.*
   (b) *Let $g : \mathbb{R} \to \mathbb{R}$ be a function such that $g(\alpha) = d_\alpha(c)$, then $\forall x \in [a,b] : \alpha x + g(\alpha) \leq f(x) \leq \alpha x + \max\{d_\alpha(a), d_\alpha(b)\}$.*
3. *If f' is continuous and monotone decreasing on $[a,b]$, we have*
   (a) *$\forall \alpha \in [f'(b), f'(a)], \exists c \in [a,b] : f'(c) = \alpha$.*
   (b) *Let $g : \mathbb{R} \to \mathbb{R}$ be a function such that $g(\alpha) = d_\alpha(c)$, then $\forall x \in [a,b] : \alpha x + \min\{d_\alpha(a), d_\alpha(b)\} \leq f(x) \leq \alpha x + g(\alpha)$.*

Proof. See the proof of Theorem 3 in [15].

To illustrate the usefulness of Theorem 3, we give the functions f' and g for some elementary functions in Table 1. Figure 2 gives a procedure to find Chebyshev affine approximation of a function $f \in \mathcal{C}^1([a,b])$ such that f' is monotone, when given the function g satisfying the conditions in Theorem 3. Noting that the exact value $\alpha^* = f'(c^*) = (f(b)-f(a))/(b-a) \leq \alpha = \lceil(\lceil f(b)\rceil - \lfloor f(a)\rfloor)/(b-a)\rceil$ for some $c^* \in [a,b]$, hence, $\alpha \geq \min\{f'(a), f'(b)\}$; we then have the proof of the procedure in Figure 2 follows Theorem 3 directly. Readers are referred to [15] for more details on this discussion.

For affine approximations of non-differentiable functions, see Section 2 of [16].

procedure `AffineApproximation`(**in** : $\hat{x}, f \in \mathcal{C}^1([a,b]), f', g$; **out** : $\alpha\hat{x} + \beta + \delta[-1,1]$)
  $f_a := \lfloor f(a) \rfloor$; $f_b := \lceil f(b) \rceil$; $\alpha := \lceil (f_b - f_a)/(b-a) \rceil$;
  if f' is monotone increasing on $[a,b]$ **then**
    $d_a := \lceil f(a) \rceil - \lfloor \alpha a \rfloor$;
    if $\alpha > \lceil f'(b) \rceil$ **then**
      $d_{min} := \lfloor f(b) \rfloor - \lceil \alpha b \rceil$; $d_{max} := d_a$;
    else
      $d_{min} := \lfloor g(\alpha) \rfloor$; $d_{max} := \max\{d_a, f_b - \lfloor \alpha b \rfloor\}$;
    end-if
  else-if f' is monotone decreasing on $[a,b]$ **then**
    $d_b := \lfloor f(b) \rfloor - \lceil \alpha b \rceil$;
    if $\alpha > \lceil f'(a) \rceil$ **then**
      $d_{min} := d_b$; $d_{max} := \lceil f(a) \rceil - \lfloor \alpha a \rfloor$;
    else
      $d_{min} := \min\{f_a - \lceil \alpha a \rceil, d_b\}$; $d_{max} := \lceil g(\alpha) \rceil$;
    end-if
  end-if
  $\beta := midpoint([d_{min}, d_{max}])$; $\delta := radius([d_{min}, d_{max}])$; // basic interval concepts
end

Fig. 2. This is a procedure to find a Chebyshev affine approximation of a function $f \in \mathcal{C}^1([a,b])$ such that f' is monotone, when given the function g in Theorem 3

Multiplication. Similar to the product of two G intervals in [5, 6], the product of two revised affine forms $\hat{x}$ and $\hat{y}$ of length n is another revised affine form $\hat{z}$ of length n defined as follows

$$z_0 = x_0 y_0 + 0.5 \sum_{i=1}^{n} x_i y_i, \;\; z_i = x_0 y_i + y_0 x_i \;\; (i = 1, \ldots, n) \tag{12}$$

$$e_z = e_x e_y + e_y \sum_{i=0}^{n} |x_i| + e_x \sum_{i=0}^{n} |y_i| + \sum_{i=1}^{n} |x_i| \sum_{i=1}^{n} |y_i| - 0.5 \sum_{i=1}^{n} |x_i y_i| \tag{13}$$

This is similar to, but tighter than, the formula for multiplication in [6] when exactly porting into revised affine form. The time complexity of (9) and (13) is $O(n)$. In rigorous computing, we use the following computations.

$$u = \lceil \sum_{i=1}^{n} |x_i| \rceil, \;\; v = \lceil \sum_{i=1}^{n} |y_i| \rceil \tag{14}$$

$$z_0 = \langle x_0 y_0 + 0.5 \sum_{i=1}^{n} x_i y_i \rangle \pm e_0, \;\; z_i = \langle x_0 y_i + y_0 x_i \rangle \pm e_i \;\; (i = 1, \ldots, n) \tag{15}$$

$$e_z = \lceil e_x e_y + e_y(|x_0| + u) + e_x(|y_0| + v) + uv + \sum_{i=0}^{n} e_i \rceil - \lfloor 0.5 \sum_{i=1}^{n} |x_i y_i| \rfloor \tag{16}$$

The multiplication defined by $\{(12), (13)\}$ or by $\{(14), (15), (16)\}$ is an inclusion function of the multiplication on $\mathbb{R}$. The proof, which can be found in [15], is based on the real expansion of xy following the definition, where $x \in \hat{x}, y \in \hat{y}$.

Division. In our implementation, we compute the quotient $\hat{z} = \hat{x}/\hat{y}$ by rewriting it as $\hat{x} \times (1/\hat{y})$. However, in [6], the author has proposed a better solution for division that has some interesting properties such as $\hat{x}/\hat{x} = 1$.

procedure NodeLevel(**in** : DAG)
  Initialize the level of each node to zero.
  for each visit in pre-order going from a parent $\mathbf{P}$ to its children $\mathbf{N}$ **do**
    $level(\mathbf{N}) := \max\{level(\mathbf{N}), level(\mathbf{P}) + 1\}$;
end

Fig. 3. This is a procedure assigning a node level to each node in a DAG.

4.2 New Combination Strategies for Constraint Propagation

In the rest, we will abuse the notions $\mathbb{I}$ and $\mathbb{A}$ to denote the real inclusion representations, $(\mathbb{I}, \mu_{\mathbb{I}})$ and $(\mathbb{A}, \mu_{\mathbb{A}})$, defined on interval arithmetic and revised affine arithmetic, respectively; where the function $\mu_{\mathbb{I}}$ is defined by (6) and the function $\mu_{\mathbb{A}}$ follows (11). In general, the performance of a propagator following the CIRD scheme depends on the design of each step in the scheme. In this section, we propose some simple strategies for each step in the CIRD scheme using the two inclusion representations, $\mathbb{I}$ and $\mathbb{A}$. Combining different strategies at all the steps makes different combination strategies for constraint propagation.

Step 1a: Initial Node Evaluation. A post-order visiting or a recursive evaluation starting from the ground node is an option for the visit at Step 1a.

Step 2a: Get the Next Node. At first, we assign a *node level* to each node in the DAG representing the constraint system such that each ancestor has a lower level than that of their descendants, hence, an ordering in Theorem 1 can be obtained easily. Figure 3 gives a simple procedure for this purpose. $\mathcal{L}_p$ is sorted in the ascending order of node levels. It is to maintain that ancestors being taken into pruning processes before their descendants. $\mathcal{L}_e$ is sorted in the descending order of node levels. It is to sure that descendants being evaluated before their ancestors. There are two simple strategies to get the next node from $\{\mathcal{L}_e, \mathcal{L}_p\}$. The first one is to get the next node from $\mathcal{L}_p$ whenever it is not empty. The second one is to get the next node from one of the two waiting lists until it becomes empty, then switch to the other list. In our implementation, we use the first simple strategy. More complicated strategies for choosing the next node can be used as alternatives, for example, based on the pruning efficiency of nodes.

Step 2b: Node Evaluation. For the node evaluation at each node $\mathbf{N}$, we can perform $\texttt{NEV}(\mathbf{N}, \mathbb{A})$ and $\texttt{NEV}(\mathbf{N}, \mathbb{I})$ in any order, if $\mathbf{N}$ is not the ground node. At Step 2(b)ii, Step 2(b)iii and Step 2(c)iii, we only count on the changes of $\tau(\mathbf{N})$ in our current implementation. A change of $\tau(\mathbf{N})$ is often considered enough if the ratio of the new width to the old width is less than a number predefined $r_w \in (0, 1)$ and the difference between the old width and the new width is greater than a predefined number $d_w > 0$ (see [1] for details). More complicated criteria that have been used in constraint programming can be used as alternatives.

Step 2c: Node Pruning. The subset $\mathcal{T}$ at this step can be chosen as $\{\mathbb{I}, \mathbb{A}\}$. For node pruning, we use $\texttt{PCS}(\mathbf{N}, \{\mathbb{I}\})$ and the following subsystem of $\texttt{PCS}(\mathbf{N}, \{\mathbb{A}\})$:

$$\texttt{PCS}_L(\mathbf{N}, \{\mathbb{A}\}) \equiv \begin{cases} \{\bigwedge_{i=1}^{k} \texttt{ICS}(\mathbb{A}(\mathbf{C}_i), \tau(\mathbf{C}_i))\} & \text{if } \mathbf{N} \text{ is ground,} \\ \{\texttt{ICS}(\mathbb{A}(\mathbf{N}), \tau(\mathbf{N})) \wedge \bigwedge_{i=1}^{k} \texttt{ICS}(\mathbb{A}(\mathbf{C}_i), \tau(\mathbf{C}_i))\} & \text{otherwise,} \end{cases}$$

$$\text{where } \mathtt{ICS}(\mathbb{A}(\mathbf{M}), D) \equiv \left\{ \begin{array}{l} x_{\mathbf{M},0} + \sum_{i=1}^{k} x_{\mathbf{M},i}\varepsilon_i + e_{\mathbf{M}}\varepsilon_{\mathbf{M}} = \vartheta_{\mathbf{M}}; \\ \varepsilon_i \in [-1,1] \ (i = 1,\ldots,n); \\ \varepsilon_{\mathbf{M}} \in [-1,1]; \vartheta_{\mathbf{M}} \in D; \end{array} \right\}$$

Example. We give the node levels for the example described in Section 2.2 in brackets next to the node names in Figure 1*b*. At the beginning, we have

$\tau(\mathbf{N}_1) = \mathbb{I}(\mathbf{N}_1) = [1,3]; \qquad \mathbb{A}(\mathbf{N}_1) = 2 + \varepsilon_1$
$\tau(\mathbf{N}_2) = \mathbb{I}(\mathbf{N}_2) = [1,9]; \qquad \mathbb{A}(\mathbf{N}_2) = 5 + 4\varepsilon_2$
$\tau(\mathbf{N}_i) = \mathbb{I}(\mathbf{N}_i) = \mathbb{A}(\mathbf{N}_i) = [-\infty, +\infty] \ (i = 3,4,5)$
$\tau(\mathbf{N}_6) = \mathbb{I}(\mathbf{N}_6) = [0,0]; \ \mathbb{A}(\mathbf{N}_6) = 0 \ ; \ \ \tau(\mathbf{N}_7) = \mathbb{I}(\mathbf{N}_7) = \mathbb{A}(\mathbf{N}_7) = [-\infty, 9]$

After the initial node evaluation, we have the following changes:

$\tau(\mathbf{N}_3) = \mathbb{I}(\mathbf{N}_3) = [1,9]; \ \ \mathbb{A}(\mathbf{N}_3) = 4.5 + 4\varepsilon_1 + 0.5[-1,1]$
$\tau(\mathbf{N}_4) = \mathbb{I}(\mathbf{N}_4) = [1,27]; \ \mathbb{A}(\mathbf{N}_4) = 10 + 5\varepsilon_1 + 8\varepsilon_2 + 4[-1,1]$
$\tau(\mathbf{N}_5) = \mathbb{I}(\mathbf{N}_5) = [1,3]; \ \ \mathbb{A}(\mathbf{N}_5) = 2.125 + \varepsilon_2 + 0.125[-1,1]$
$\tau(\mathbf{N}_6) = \mathbb{I}(\mathbf{N}_6) = [0,0]; \ \ \mathbb{A}(\mathbf{N}_6) = -13.375 - 6\varepsilon_1 - 15\varepsilon_2 + 8.625[-1,1]$
$\tau(\mathbf{N}_7) = \mathbb{I}(\mathbf{N}_7) = [9,9]; \ \ \mathbb{A}(\mathbf{N}_7) = 42.25 + 19\varepsilon_1 + 26\varepsilon_2 + 12.25[-1,1]$

Denoting the variable $\vartheta_{\mathbf{N}_i}$ by v_i, we have, for example, $\mathtt{PCS}(\mathbf{N}_6, \{\mathbb{A}\}) \equiv \mathtt{PCS}(\mathbf{N}_6, \{\mathbb{I}\}) \wedge \mathtt{PCS}_L(\mathbf{N}_6, \{\mathbb{A}\})$, where $\mathtt{PCS}_L(\mathbf{N}_6, \{\mathbb{A}\}) \equiv \{4.5 + 4\varepsilon_1 + 0.5\varepsilon_{\mathbf{N}_3} = v_3; \ 10 + 5\varepsilon_1 + 8\varepsilon_2 + 4\varepsilon_{\mathbf{N}_4} = v_4; \ 2.125 + \varepsilon_2 + 0.125\varepsilon_{\mathbf{N}_5} = v_5; \ -13.375 - 6\varepsilon_1 - 15\varepsilon_2 + 8.625\varepsilon_{\mathbf{N}_6} = v_6 \mid (\varepsilon_1, \varepsilon_2, \varepsilon_{\mathbf{N}_3}, \varepsilon_{\mathbf{N}_4}, \varepsilon_{\mathbf{N}_5}, \varepsilon_{\mathbf{N}_6}) \in [-1,1]^6; \ v_3 \in [1,9]; \ v_4 \in [1,27]; \ v_5 \in [1,3]; \ v_6 \in [0,0]\}$, and $\mathtt{PCS}(\mathbf{N}_6, \{\mathbb{I}\}) \equiv \{v_3 - 2v_4 + v_5 = v_6 \mid \ v_3 \in [1,9]; \ v_4 \in [1,27]; \ v_5 \in [1,3]; \ v_6 \in [0,0]\}$.

The combination of the following backward propagation and affine pruning techniques makes different strategies for node pruning in the `CIRD` scheme.

Backward Propagation. If $\mathbf{N}$ is not the ground, the domains of the variables of the constraint system $\mathtt{PCS}(\mathbf{N}, \{\mathbb{I}\})$ can be pruned by a pruning technique which is called *backward propagation* in [1, 14]. In brief, let f be the elementary operation represented by a node $\mathbf{N}$, we then have the relation $\vartheta_{\mathbf{N}} = f(\{\vartheta_{\mathbf{C}_i}\}_{i=1}^k)$. For each i in $\{1,\ldots,k\}$, the backward propagation computes a cheap evaluation of the i-th projection of the relation $\vartheta_{\mathbf{N}} = f(\{\vartheta_{\mathbf{C}_i}\}_{i=1}^k)$ onto $\vartheta_{\mathbf{C}_i}$. In case there exist a function $g_i : \mathbb{R}^k \to \mathbb{R}$ such that we can write $\vartheta_{\mathbf{C}_i} = g_i(\vartheta_{\mathbf{N}}, \{\vartheta_{\mathbf{C}_j}\}_{j=1;j\neq i}^k)$. Let G_i be an inclusion function of g_i in $\mathbb{I}$. The i-th backward propagation at $\mathbf{N}$ is then defined by

$$\mathbb{I}(\mathbf{C}_i) := \mathbb{I}(\mathbf{C}_i) \cap G_i(\mathbb{I}(\mathbf{N}), \{\mathbb{I}(\mathbf{C}_j)\}_{j=1;j\neq i}^k) \ (i = 1,\ldots,k) \tag{17}$$

The **other cases** are described in detail in [1, 14]. After the backward propagation, at Step 2(c)iii we only need to consider k nodes $\mathcal{H} = \{\mathbf{C}_i \mid i = 1,\ldots,k\}$ for update and for putting into the waiting lists $\mathcal{L}_e$ and $\mathcal{L}_p$.

Affine Pruning. In $\mathbb{A}$, each variable of the input constraint system is associated with one noise symbol ε_i $(i = 1,\ldots,n)$. The system $\mathtt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$ is a linear constraint system, therefore, the domains of the variables of $\mathtt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$ can be pruned by using a *safe* linear programming technique [17]. If the operation represented by $\mathbf{N}$ is linear, we can apply a *safe* linear programming technique to $\mathtt{PCS}(\mathbf{N}, \{\mathbb{A}\})$, instead of $\mathtt{PCS}_L(\mathbf{N}, \{\mathbb{A}\})$, to get tighter bounds on the variables. For efficiency, only the domains of the variables $\{\vartheta_{\mathbf{C}_i}\}_{i=1}^k$ and/or $\{\varepsilon_i\}_{i=1}^n$ are

needed to be pruned. We can devise three possible pruning strategies for Step 2(c)iii. The first strategy only requires to prune the domains of $\{\vartheta_{\mathbf{C}_i}\}_{i=1}^k$, after that, considers the update for $\mathcal{H} = \{\mathbf{C}_i\}_{i=1}^k$. The second strategy only requires to prune the domains of $\{\varepsilon_i\}_{i=1}^n$. The third strategy is to prune the domains of both $\{\vartheta_{\mathbf{C}_i}\}_{i=1}^k$ and $\{\varepsilon_i\}_{i=1}^n$. For the last two strategies, the set $\mathcal{H}$ can be chosen as any subset of the set of $\mathbf{N}$'s descendants whose noise variables in $\mu_{\mathbb{A}}$ have just been pruned. In our implementation, we use the second pruning strategy with two options for $\mathcal{H}$: the set of $\mathbf{N}$'s descendants or the set of variables associated with ε_i $(i = 1, \ldots, n)$. If for each $i \in \{1, \ldots, n\}$ the new domain of noise variable ε_i is $[a_i, b_i] \subseteq [-1, 1]$, then the range update at $\mathbf{M} \in \mathcal{H}$ will be

$$\tau(\mathbf{M}) := \tau(\mathbf{M}) \cap (x_{\mathbf{M},0} + \sum_{i=1}^{n} x_{\mathbf{M},i}[a_i, b_i] + e_{\mathbf{M}}[-1, 1]) \tag{18}$$

Remark 1. The cost of linear programming is high, therefore, we should use the affine pruning technique only if the pruning ratio is high. We propose to use the affine pruning technique only at low level nodes and only if the accumulative error $e_{\mathbf{M}}$ of each node $\mathbf{M}$ involving the above linear systems is small enough. That is, the range of the operation at $\mathbf{M}$ lies in a thin slot between two hyperplanes $x_{\mathbf{M},0}+\sum_{i=1}^n x_{\mathbf{M},i}\varepsilon_i - e_{\mathbf{M}}$ and $x_{\mathbf{M},0}+\sum_{i=1}^n x_{\mathbf{M},i}\varepsilon_i + e_{\mathbf{M}}$ in the space of $(\varepsilon_1, \ldots, \varepsilon_n)$.

5 Experiments

Preliminary Comparisons with Linear Relaxation based Techniques. We first compare the proposed technique with a recent mathematical solving technique, called `A2`, in [6] which was specially designed to solve nonlinear equation systems. The `A2` algorithm converts the equation system into *separable form*, and then uses affine arithmetic to enclose the system by a linear relaxation system $\{\mathbf{L}(x, y) = Ax + By + b,\ x \in \mathbf{x},\ y \in \mathbf{y}\}$; where A and B are real matrices, b is a real vector, and $\mathbf{x}$ and $\mathbf{y}$ are interval vectors. This technique has to assume a posterior-condition that A is invertible in order to use the reduction rule. No rigorous rounding technique is found in [6]. We take the example used for illustrating the power of the `A2` algorithm in [6] for the comparison:

$$\begin{cases} ((4x_3 + 3x_6)x_3 + 2x_5)x_3 + x_4 = 0,\ ((4x_2 + 3x_6)x_2 + 2x_5)x_2 + x_4 = 0 \\ ((4x_1 + 3x_6)x_1 + 2x_5)x_1 + x_4 = 0,\ x_4 + x_5 + x_6 + 1 = 0 \\ (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_2 + (((x_3 + x_6)x_3 + x_5)x_3 + x_4)x_3 = 0 \\ (((x_1 + x_6)x_1 + x_5)x_1 + x_4)x_1 + (((x_2 + x_6)x_2 + x_5)x_2 + x_4)x_3 = 0 \\ x_1 \in [0.0333, 0.2173],\ x_2 \in [0.4000, 0.6000],\ x_3 \in [0.7826, 0.9666] \\ x_4 \in [-0.3071, -0.1071],\ x_5 \in [1.1071, 1.3071],\ x_6 \in [-2.1000, -1.9000] \end{cases} \tag{19}$$

The system (19) is known to have an unique solution. To solve (19) on a 1.7 GHz Pentium PC at the resolution 10^{-5} using a bisection search; `A2` has to perform 917 splittings in 3.46 seconds to reduce the problem to 5 boxes (see [6]); while an instance, called `CIRD1`,[4] of the `CIRD` scheme performs 54 splittings in only 0.118 seconds to reduce the problem to 3 boxes. Hence, `CIRD1` is about 29.3 times

[4] In this paper we use a new implementation of `CIRD1`, which is an improvement of the old version used in [15].

Table 2. Comparison between `Quad` and `CIRD1`: time is in seconds; $\#It$ is the number of splittings; $\#Box$ is the number of boxes in output. The cells are filled with "n/a" if results are not yet available for comparison in this submission, due to our limited access to the code of `Quad`.

	Quad				CIRD1				Time Ratio
Problem	$\#It$	$\#Box$	Time (sec.)	CPU speed	$\#It$	$\#Box$	Time (sec.)	CPU speed	Quad/CIRD1
Gough-Steward [$n = 9$]	24	4	183.0	1.0 GHz	912	4	2.7	1.7 GHz	39.9
Yama196 [$n = 30$]	108	16	31.4	2.66 GHz	25	2	3.8	1.7 GHz	12.9
Yama196 [$n = 60$]	n/a	n/a	n/a	n/a	18	2	21.0	1.7 GHz	
Yama196 [$n = 100$]	n/a	n/a	n/a	n/a	20	2	85.8	1.7 GHz	
Yama196 [$n = 200$]	n/a	n/a	n/a	n/a	19	2	560.2	1.7 GHz	
Yama196 [$n = 300$]	n/a	n/a	n/a	n/a	20	2	1878.1	1.7 GHz	

faster than `A2` for the system (19), while it is more rigorous and accurate than `A2`. Another technique to compare with is a very recent filtering technique called `Quad` in [3], which was specifically designed to address quadratic constraints and an extension of `Quad` in [4]. We take two problems, *Gough-Steward* and *Yama196*, from [3] and [4] respectively for comparison. *Gough-Steward* is a 9-dimensional quadratic equation system in Robotics having four solutions [3]. *Yama196* is a series of high dimensional problems consisting of n variables and n equations of form $\{(n+1)^2 x_{i-1} - 2(n+1)^2 x_i + (n+1)^2 x_{i+1} + e^{x_i} = 0,\ x_i \in [-10, 10] \mid i = 1, \ldots, n\}$, where $x_0 = x_{n+1} = 0$. Similar to [4], we use the resolution 10^{-8} for these problems. Table 2 gives the comparison between `CIRD1` and `Quad` (at the same resolution) for the above two problems whose results has been reported in [3, 4].

Comparison with Interval Propagation Techniques. We have carried out experiments on `CIRD1` and two other recent interval propagation techniques. The first one is the Box Consistency in ILOG Solver 6.0, denoted by `BOX`. The second one is called `HC4` (Revised Hull Consistency) in [1]. The experiments are carried out on 33 problems which are *unbiasedly* selected and divided into 5 test cases. The test case T_1 consists of 8 problems with isolated solutions that are solvable by all three propagators. The test case T_2 consists of 4 problems with isolated solutions that are solvable by only two propagators `CIRD1` and `BOX`. The test case T_3 consists of 8 problems with isolated solutions that cause at least two of three techniques being stopped due to timeout or due to running more than 10^6 splittings. The test case T_4 consists of 7 small problems with continuum of solutions that are solvable at resolution 10^{-2}. The test case T_5 consists of 6 hard problems with continuum of solutions that are solvable at resolution 10^{-1}. The timeout value is **10 hours** for all the test cases, it will be used as the running time for the techniques which is timeout in the next result analysis (i.e. we are in favor of slow techniques). For the first three test cases, the resolution is 10^{-4} and the search to be used is bisection. For the last two test cases, the search to be used is a simple search technique, called `UCA6`, for inequalities (see [7, 8]). The comparison of the interval propagation techniques is based on the measures of

1. *The running time:* The relative ratio of the running time of each propagator to that of `CIRD1` is called the *relative time ratio*.
2. *The number of boxes:* The relative ratio of that number of boxes in the output of each propagator to that of `CIRD1` is called the *relative cluster ratio*.
3. *The number of splittings:* the number of splittings in search needed to solve the problems. The relative ratio of the number of splittings used by each propagator to that of `CIRD1` is called the *relative iteration ratio*.

Table 3. (*a*) The average of the relative time ratios is taken over all the problems in the test cases T_1, T_2, T_3; the averages of the other relative ratios are taken over the problems in the test case T_1, i.e. over the problems which are solvable by all the techniques. (*b*) The averages of the relative ratios are taken over all the problems in the test cases T_4, T_5. In general, the lower the relative ratio, the better the performance/quality; and the higher the inner volume ratio, the better the quality.

Propagator	(*a*) *Isolated Solutions*				(*b*) *Continuum of Solutions*			
	Relative time ratio	Relative reduction ratio	Relative cluster ratio	Relative iteration ratio	Relative time ratio	Inner volume ratio	Relative cluster ratio	Relative iteration ratio
`CIRD1`	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.945**	**1.000**	**1.000**
`BOX`	1429.660	5.323	30.206	4.263	3.414	0.944	1.102	1.056
`HC4`	17283.614	7.722	105.825	5.515	60.101	0.941	1.168	1.118

Table 4. This table contains the averages of the relative time ratios taken over the problems in each test case.

Propagator	(*a*) *Isolated Solutions*			(*b*) *Continuum of Solutions*	
	Test case T_1	Test case T_2	Test case T_3	Test case T_4	Test case T_5
`CIRD1`	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
`BOX`	8.33	6097.45	517.10	2.33	4.68
`HC4`	54.47	83009.81	1649.66	31.42	93.56

4. *The volume of boxes (only for T_1, T_2, T_3):* We consider the reduction per dimension when replacing the set of output boxes by a volume-equivalent hypercube. The relative ratio of the reduction gained by each propagator to that of `CIRD1` is called the *relative reduction ratio*.
5. *The volume of inner boxes (only for T_4, T_5):* The ratio of the volume of inner boxes to the volume of all output boxes is called the *inner volume ratio*.

The overviews of results in our experiments are given in Table 3 and Table 4. Clearly, `CIRD1` is superior than `BOX` and `HC4` in performance and quality for the problems with isolated solutions. `CIRD1` still outperforms the others for the problems with continuum of solutions while being a little better than the others in quality of the output.

Remark 2. We have also carried out experiments on the naive use of affine arithmetic as a replacement of interval arithmetic in interval constraint propagation. However, the performance of obtained techniques is even worse than the one using interval arithmetic. Lack of space does not allows showing the results here.

6 Conclusion

In this paper, we propose a novel generic scheme, `CIRD`, for constraint propagation using different inclusion representations on DAG. The scheme is applicable to most of known inclusion representations, including interval arithmetic, affine arithmetic, polyhedral/quadratic enclosures and their generalizations. The modifications and improvements on the rigorous computations of affine arithmetic are also proposed. As a result, we give several new combination strategies for constraint propagation based on interval arithmetic, affine arithmetic, interval constraint propagation and (safe) linear programming. After all, we show by experiments that one implementation, `CIRD1`, outperforms recent techniques by 1-4 orders of magnitude in speed, while still being better in quality measures.

Acknowledgements. We would like to thank ILOG for the licenses of ILOG Solver/CPLEX used in the European COCONUT project (IST-2000-26063), and thank the other COCONUT partners for related materials.

References

1. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.F.: Revising Hull and Box Consistency. In: Proceedings of the International Conference on Logic Programming (ICLP'99), Las Cruces, USA (1999) 230–244
2. Jaulin, L., Kieffer, M., Didrit, O., Walter, E.: Applied Interval Analysis. First edn. Springer (2001)
3. Lebbah, Y., Rueher, M., Michel, C.: A Global Filtering Algorithm for Handling Systems of Quadratic Equations and Inequations. In: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'2003). Volume LNCS 2470., Springer (2003) 109–123
4. Lebbah, Y., Michel, C., Rueher, M.: Global Filtering Algorithms Based on Linear Relaxations . In: Notes of the 2nd International Workshop on Global Constrained Optimization and Constraint Satisfaction (COCOS 2003), Switzerland (2003)
5. Kolev, L.V.: Automatic Computation of a Linear Interval Enclosure. Realiable Computing **7** (2001) 17–18
6. Kolev, L.V.: An Improved Interval Linearization for Solving Non-Linear Problems. In: 10th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN2002), France (2002)
7. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Search Techniques for Non-linear CSPs with Inequalities. In: Proceedings of the 14th Canadian Conference on Artificial Intelligence. (2001)
8. Vu, X.H., Sam-Haroud, D., Silaghi, M.C.: Numerical Constraint Satisfaction Problems with Non-isolated Solutions. In: Global Optimization and Constraint Satisfaction. Volume LNCS 2861., Springer-Verlag (2003) 194–210
9. Hickey, T.J., Ju, Q., Van Emden, M.H.: Interval Arithmetic: from Principles to Implementation. Journal of the ACM (JACM) **48(5)** (2001) 1038–1068
10. Comba, J.L.D., Stolfi, J.: Affine Arithmetic and its Applications to Computer Graphics. In: Proceedings of SIBGRAPI'93, Brazil (1993)
11. Stolfi, J., de Figueiredo, L.H.: Self-Validated Numerical Methods and Applications. In: Monograph for 21st Brazilian Mathematics Colloquium (IMPA), Brazil (1997)
12. Messine, F.: Extentions of Affine Arithmetic: Application to Unconstrained Global Optimization. Journal of Universal Computer Science **8(11)** (2002) 992–1015
13. Martin, R., Shou, H., Voiculescu, I., Bowyer, A., Wang, G.: Comparison of Interval Methods for Plotting Algebraic Curves. Computer Aided Geometric Design **19(7)** (2002) 553–587
14. Schichl, H., Neumaier, A.: Interval Analysis on Directed Acyclic Graphs for Global Optimization (2004) preprint - University of Vienna, Autria.
15. Vu, X.H., Sam-Haroud, D., Faltings, B.: A Generic Scheme for Combining Multiple Inclusion Representations in Numerical Constraint Propagation. Technical Report IC/2004/39, Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland (2004)
16. Kolev, L.V.: A New Method for Global Solution of Systems of Non-Linear Equations. Realiable Computing **4** (1998) 125–146
17. Neumaier, A., Shcherbina, O.: Safe Bounds in Linear and Mixed-Integer Programming. Mathematical Programming A **99** (2004) 283–296

The Optimistic Principle and Optimistic Pruning: A Preliminary Report *

Eugene C. Freuder and Yuanlin Zhang**

Cork Constraint Computation Centre,
University College Cork, Ireland
{e.freuder, y.zhang}@4c.ucc.ie

Abstract. In contrast to the deterministic and sound inference during search, we propose to make inference in an optimistic way, to speed up the solving of problems. Specifically, we examine optimistic pruning where a value is excluded from consideration when it is close to arc inconsistent. Our preliminary empirical study shows that under a proper level of optimism, optimistic pruning improves the performance of a MAC algorithm on the hard random problem instances close to (from the satisfiable side of) the phase transition point.

1 introduction

Much human and computer search simplifies the search space by making assumptions, which can later be incrementally or totally retracted if they eliminate all satisfactory solutions. Basic backtrack search, which underlies much of CSP solving, can be viewed in these terms. However, there the expectation is that failure is likely and backtracking inevitable. Here we propose an important 'psychological shift'. Rather than take the natural, conservative attitude 'we can't do that, it could lose solutions', we propose a more 'optimistic' approach: 'let's try that, it might work'. Once we have done this basic bit of 'lateral thinking' a whole new world of possibilities opens up to us.

More specifically, we can consider situations in which conditions C allow us to conclude a useful property P, and ask: Suppose we can come 'close' to establishing C, might it prove profitable to assume P anyway? More specifically still, we can consider properties like arc-inconsistency that allows us to prune values from variable domains and ask: Suppose a value is 'close' to being arc inconsistent, might it be profitable to assume it is? At worst, we might prune values that leave us without a solution, or without the best solution if we are optimizing; but we can always 'go back' and undo our assumption if need be. The question, as with all heuristic methods, is whether the potential gains outweigh the potential losses.

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

** Current address: Department of Computer Science, Texas Tech University, Lubbock, Texas.

It is actually rather curious that a field that emphasizes backtracking in search more than any other has not explored more backtracking in inference. Of course, choice and backtrack is forced on us for search. Inference is attractive in that it can be deterministic. However, embracing the risk involved in voluntary choice, backstopped if necessary by backtracking, may prove powerful as well. Stochastic approaches, such as local search and random restart, have demonstrated that taking risks can be fruitful.

2 Optimistic pruning

Many methods have been developed to prune the search space. We restrict ourselves to a specific and well studied pruning process: maintaining arc consistency during search (MAC) [3]. In an optimistic MAC, a value is excluded from further consideration if it is *close* to be arc inconsistent with respect to a constraint. The optimism here is that if a value is close to be arc inconsistent, it stands a very small chance to be supported in the future. How do we measure the closeness of arc consistency?

Value ordering heuristics are frequently used to improve the efficiency of solving a CSP problem. When instantiating a variable, a more promising value is tried first. In other words, the less promising values are not likely to be a correct assignment for the current variable. In optimistic MAC, the closeness to arc inconsistency can be measured by any value ordering heuristic, and the less promising a value is, the closer it is to arc inconsistency. When instantiating a variable, it is cheap to calculate an ordering on its values, but it could be expensive to maintain a value ordering on *all* variables during each invocation of arc consistency. Here, we propose two ways to determine whether a value is close to be arc inconsistent.

The first is to check the number of supports of a value with respect to each constraint on it. When this number falls below a threshold, the value will be removed from further consideration. The second depends on the proportion of supports that a value has lost during a search procedure. If the proportion drops below a certain percentage, the value will be deleted optimistically because it loses supports "faster" than the other values.

Example Consider three variables x, y and z that can take values from a domain of $\{-5, -2, -1, 1, 2, 5\}$. The constraint between x and y is $|x| = |y|$ when $x = \pm 5$ and otherwise is almost "$|x| = |y| + 1$ or $|y| = |x| + 1$" (except for -2 of x and -1 of y) as shown in Fig. 1. The other constraints are $|x| = |z|$ and $|y| = |z|$. Since -2 of x has only one support in y, it can be removed optimistically, resulting in the removal of $\{-1, 1, -2, 2\}$ from the domains of x, y and z. Now, it is easy to find a solution for the problem.

In our experiments, the optimism defined above leads to too many values to be deleted, resulting in no solution for many originally satisfiable problem instances. As more variables are instantiated, the condition of optimism is easier to be satisfied in the later stage of the search. To curb this tendency, one method

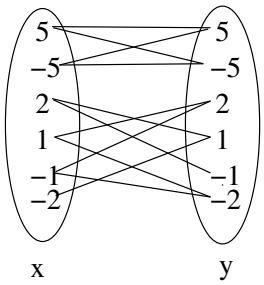


Fig. 1. A constraint between variables x and y

is to turn off the optimism at certain depth (i.e., the number of instantiated variables) of search.

2.1 Optimistic pruning algorithm

In this section, we present an optimistic MAC algorithm that removes a value optimistically in terms of the number of its supports. In the algorithm we need to maintain the number of supports of a value, implying that the techniques developed in AC-4 [2] are a good choice for this optimistic algorithm.

For each value with respect to a constraint incident on it, AC-4 maintains not only the number of, but also a list of, all supports for the value. At the initialization stage of AC-4, all the values that have zero support will be put into a queue so that later we are able to propagate the removal of them to all their supports. Specifically, when a value is taken from the queue, for each of its supports a, decrease the number of supports of a by one. The optimism comes in now. When a's number of supports falls below a threshold, a will be put into the queue, waiting to be deleted. At the same time, we can also check whether we are beyond certain depth of search and if so turn off the optimism. The propagation continues until no value is left in the queue.

The kernel of a preprocessing AC algorithm or MAC [3] is the propagation algorithm. Given a queue of removed values, the propagation algorithm of the optimistic MAC, listed in Fig. 2, popagates the values optimistically. The algorithm needs the following data structures. For any value a of a variable i and a constraint between i and j, `counter(i,a,j)` and `supports(i,a,j)` are the number, and the list respectively, of all supports of a with respect to the constraint between i and j. A list of values to be removed, each of which is denoted by `(i,b)` (a value b of variable i), are put in the queue `Q`.

The algorithm opti-propogate(`Q`) in Fig. 2 propogates the deletion of the values in `Q` and removes an affected value optimistically in terms of the optimistic condition implemented by the procdure removable-optimistically(`j, a, i`).

Line 1 in Fig. 3 ignores the optimism when the variable i is instantiated or has only one value left in its domain. In this case, any value of j has only one

```
algorithm opti-propogate(Queue Q)
    begin
        while Q not empty do
            select and delete a value (i,b) from Q;
            for each neighboring variable j of i do
                for each value a in supports(i,b,j)
                    if removable-optimistically( j, a, i)
                        delete (i,a);
                        Q ← Q ∪ { (i,a) };
                    endif
                endfor
            endfor
        endwhile
    end
```

Fig. 2. The AC-4 propagation algorithm

```
procedure removable-optimistically( j, a, i)
    begin
        counter(j,a,i) = counter(j,a,i)-1;
        if counter(j,a,i) is zero
            return true;
1.      if domain of i has only one value left
            return false;
2.      if counter(j,a,i) is smaller than a threshold
            and the current search depth is shallower than a threshold
                return true;
        else return false
    end
```

Fig. 3. procedure to check whether a value is optimistically removable

support in i and the optimism is turned off because otherwise all values of j will be excluded optimistically, resulting in a search failure. Line 2 is to apply the optimistic removal criteria to the value a of variable j.

3 Experimental results

Experiments are designed to examine the effectiveness of the optimistic MAC and to characterise the problems on which optimistic pruning is effective. Uni-

formly randomized binary constraint satisfaction problems (based on model B) serve these purposes well. To specify a set of problem instances, we need the parameters of the number of variables n, the maximum domain size d of the variables, the number of constraints e, and the tightness of the constraints t (t is the number of disallowed tuples). A tuple $< n, d, e, t >$ is used to distinguish different classes of problem instances.

In our experiments, $< n, d, e, t >$ is set in terms of the following rules. n, d are chosen arbitrarily and independently. To locate the hardest problems, the number of constraints is set to be about 93% of $n(n-1)/2$, the number of all possible constraints. Once n, d, e are fixed, we locate the t at the peak of the phase transition area. From the instances of $< n, d, e, t >$ we choose only those instances that are satisfiable. The reason to do so is to locate the problems where optimistic pruning is promising. For unsatisfiable instances, due to the incompleteness of optimistic pruning, finally we have to resort to a complete search algorithm to prove the unsatisfiability and thus optimistic pruning could not improve the performance of the hosting search algorithm.

After $< n, d, e, t >$ is fixed, we vary e, the number of constraints, to generate a sequence of classes of instances. In this way, we have hard problems in a relatively large range of varying e's, in contrast to generating instances by varying t with n, d, e fixed.

For example, after setting n and d to be 30 and 10 respectively, e should be 405 (about 93% of all constraints). Through experiments on various t's, we find the phase transition point where $t = 15$. With n, d, t being 30, 10 and 15 respectively, experimental data shown in Fig. 4 (The diagram is colorful and best viewed on a computer) are collected by setting e to be various values less than 405.

The optimistic pruning algorithms are parameterized by the minimum number of supports a value should have with respect to any incident constraints, and the *threshold depth* that is the search depth where the optimism is turned off. In all the experiments reported here, we set the minimum support to be 2, and vary the threshold depth from 2 to 4. In Fig. 4, optiMAC$< 1, 2 >$ means an algorithm that removes a value if it has at most 1 support and employs a threshold depth of 2. From this figure, it is observed that the optimistic algorithm with threshold depth of 4 performs better than the non-optimistic MAC, especially on harder problems close to the phase transition point.

To see whether this observation is applicable to other classes of random problems, we explored the problems whose number of variables and domain size are around 30 and 10 respectively. Fig. 5 shows the results on $n = 27$ and d from 9 to 13. The x axis is the sequence number of different settings and y axis is the average number of constraint checks used for solving the instances in each setting. The diagram contains five components. The leftmost is for $d = 9$, the second for $d = 10$, and so on. The setting for each component varies only on the numbers of constraints. For example, the left most component is obtained by varying the number of constraints from 309 to 339 when n, d, t are 27, 8 and 12 respectively. It can be regarded as the miniaturized version of a diagram like

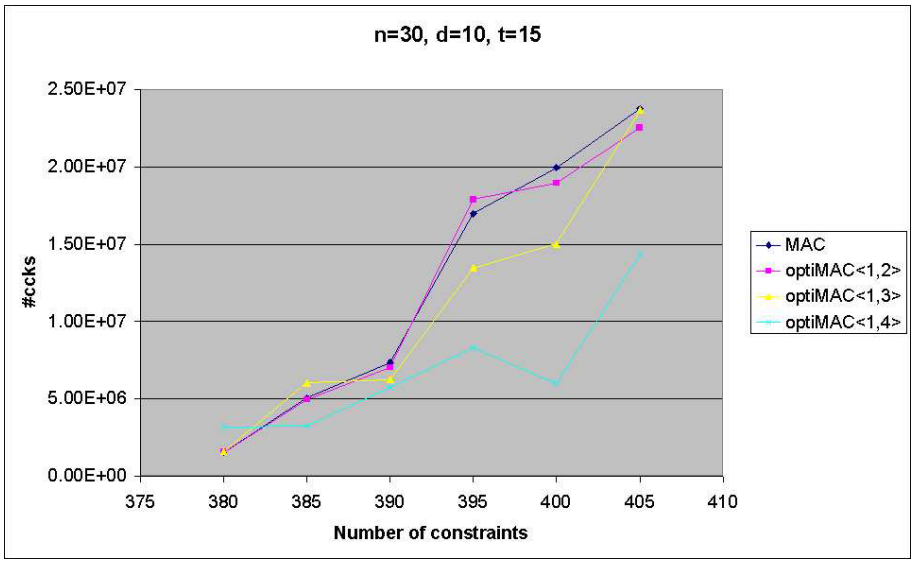


Fig. 4. Experimental results on problem instances

Fig. 4. Results on other settings are shown in Fig. 6-8 (they are colorful picures and best viewed on a computer) that use the same legends given in Fig. 4.

From these results, we can see that under a proper threshold depth, the search algorithm with optimistic pruning performs better in most cases than non-optimistic algorithm on the problems close to the peak of the phase transition area.

There are a few remarks on the data shown in the diagrams. First, the number of settings in each component varies. This is due to the fact that we eliminate settings where there exist unsatisfiable instances. This could be remedied in the future experiments by selecting a slightly smaller tightness for those settings generating unsatisfiable instances. Second, each component is supposed to be increasing when the number of constraints increases. The reason this is not true for our data might be that we did not use a sufficient number of instances for each setting $< n, d, e, t >$. (Due to the large number of settings, we test only 5 instances for each setting. We believe larger number of instances could improve the situation.)

4 Discussion

Iterative broadening reported in [1] is a search scheme under which for each variable, there is only a fixed number of values will be tried and backtracking occurs if these values fail to be consistently extensible to a solution. In this scheme, no values in the domains of *future* variables will be excluded from consideration (if

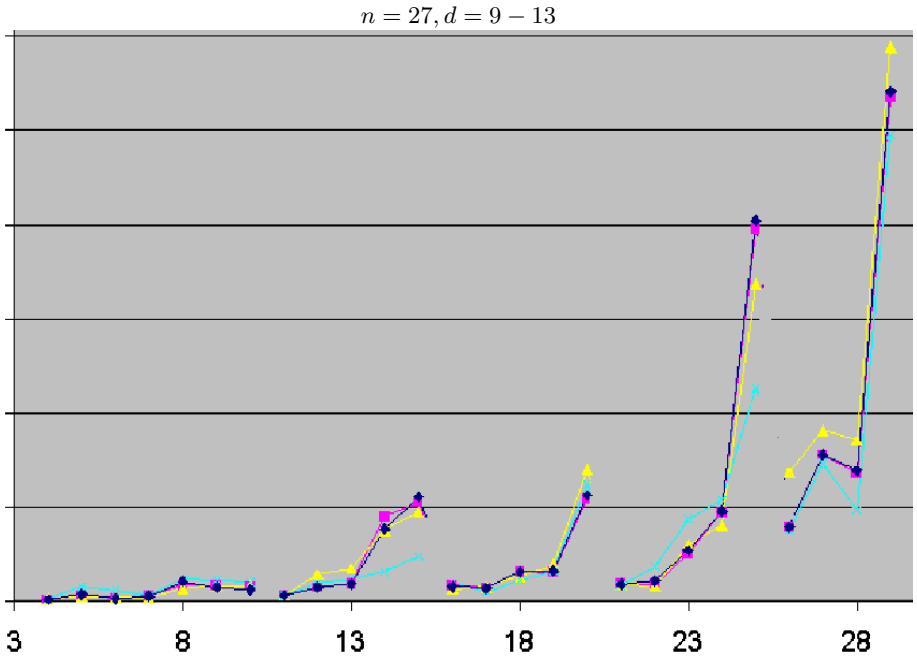


Fig. 5. Experimental results on more problem instances

they have a support with respect to every incident constraint). In the case of optimistic pruning, some values of future variables will be deleted optimistically in terms of the search depth and the number of its supports. This leads to more pruning than the iterative broadening scheme.

Some readers might have realized that the optimism MAC, especially through our experiments, is very coarse grained. For example, the optimism is turned off at a very shallow depth of 4. We had tried to increase the threshold depth of the optimistic pruning but obtained answers of unsatisfiability for some originally satisfiable instances. It seems necessary to develop more fine-grained optimism to achieve better performance. For example, when considering removing a value optimistically, we could use the information on its supports with respect to all incident constraints, rather than one constraint. In our current algorithms and implementation, the same optimism scheme is used during arc consistency after each instantiation of a variable. In this case, the optimism could be applied to the same domain repeatedly, possibly resulting in more values removed *optimistically*. To relieve this effect, we can either restrict the number of times we apply optimism to the domain of each variable in one execution of arc consistency algorithm or relate the invocation of the optimism to the size of the domain.

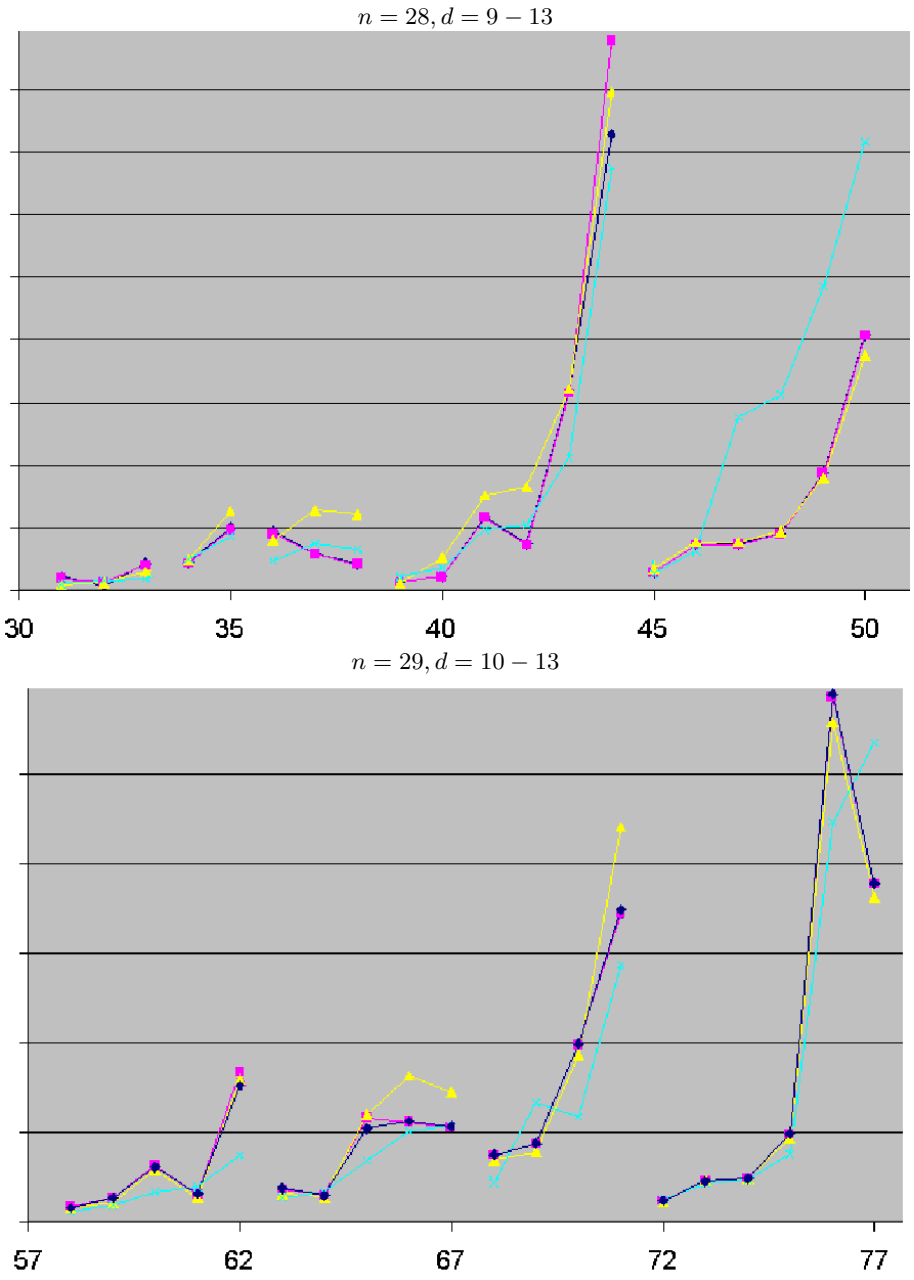


Fig. 6. Experimental results on more problem instances (continued)

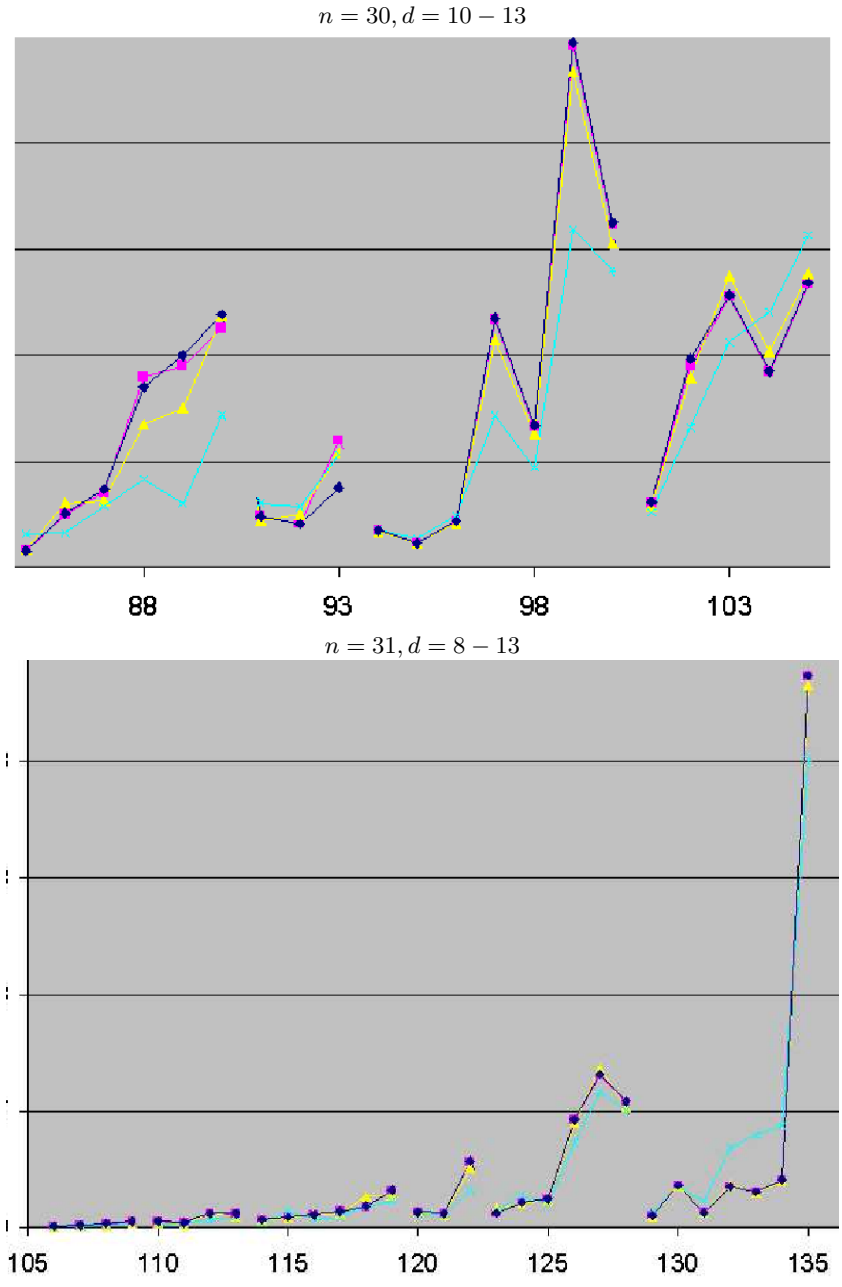
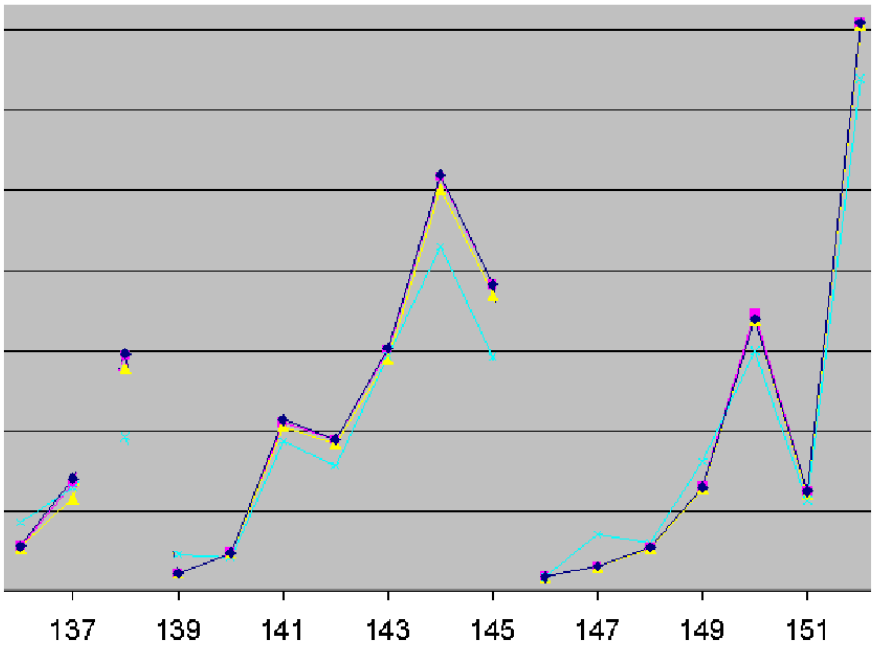


Fig. 7. Experimental results on more problem instances (continued)

$n = 32, d = 10 - 13$



$n = 33, d = 10 - 13$

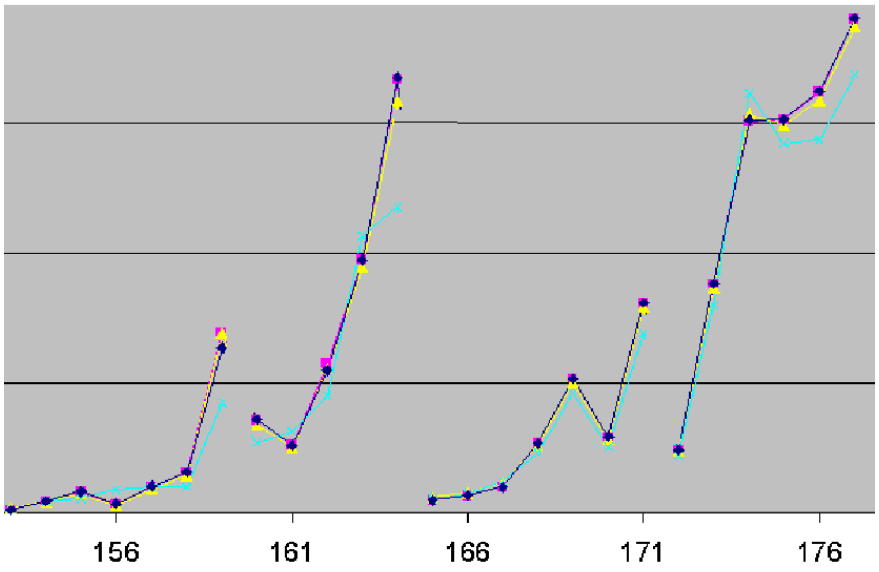


Fig. 8. Experimental results on more problem instances (continued)

5 Conclusion

We propose to make optimistic inference during search and examine a specific optimistic pruning – optimistic MAC. Preliminary experiments show that there exist problems for which optimistic pruning is very promising. With proper control of optimism (through number of supports and depth of search), this approach improves the performance of a search procedure on most problem instances close (from the satisfiable side) to the phase transition point. It is also observed that optimistic pruning makes some easy problems harder to solve. In the future we will explore the potential of optimistic pruning by designing more fine-tuned types of optimism and more importantly identifying the problems in specific application domains that can be efficiently solved by optimistic pruning.

6 Acknowledgement

We thank Barry O'Sullivan and Tudor Hulubei for many disscussions on the material presented here.

References

1. Matthew L. Ginsberg and William D. Harvey. Iterative broadening. *Artificial Intelligence*, 55(2):367–383, 1992.
2. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
3. D. Sabin and E. Freuder. Contradicting conventional wisdosdom in constraint satisfaction. In *Proceedings of the Second Workshop on Principles and Practice of Constraint Programming*, pages 10–20, Rosario, Orcas Island, Washington, 1994.