Artificial Intelligence 57 (1992) 291–321
Elsevier

A generic arc-consistency algorithm and its specializations*

Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912, USA

Yves Deville
Université Catholique de Louvain, Pl. Ste Barbe 2, B-1348 Louvain-La-Neuve, Belgium

Choh-Man Teng
Brown University, Box 1910, Providence, RI 02912, USA

Received December 1991
Revised April 1992

Abstract

Van Hentenryck, P., Y. Deville and C.-M. Teng, A generic arc-consistency algorithm and its specializations, Artificial Intelligence 57 (1992) 291–321.

Consistency techniques have been studied extensively in the past as a way of tackling constraint satisfaction problems (CSP). In particular, various arc-consistency algorithms have been proposed, originating from Waltz's filtering algorithm [27] and culminating in the optimal algorithm AC-4 of Mohr and Henderson [16]. AC-4 runs in $O(ed^2)$ in the worst case, where e is the number of arcs (or constraints) and d is the size of the largest domain. Being applicable to the whole class of (binary) CSP, these algorithms do not take into account the semantics of constraints.

In this paper, we present a new generic arc-consistency algorithm AC-5. This algorithm is parametrized on two specified procedures and can be instantiated to reduce to AC-3 and AC-4. More important, AC-5 can be instantiated to produce an $O(ed)$ algorithm for a number of important classes of constraints: functional, anti-functional, monotonic, and their generalization to (functional, anti-functional, and monotonic) piecewise constraints.

We also show that AC-5 has an important application in constraint logic programming over finite domains [24]. The kernel of the constraint solver for such a programming

Correspondence to: Pascal Van Hentenryck, Brown University, Box 1910, Providence, RI 02912, USA.

* This paper is an extended version of [3]. Originally, it was submitted as a Research Note.

0004-3702/92/$05.00 © 1992 — Elsevier Science Publishers B.V. All rights reserved

language is an arc-consistency algorithm for a set of basic constraints. We prove that AC-5, in conjunction with node consistency, provides a decision procedure for these constraints running in time $O(ed)$.

1. Introduction

Many important problems in areas like artificial intelligence, operations research, and hardware design can be viewed as constraint satisfaction problems (CSP). A CSP is defined by a finite set of variables taking values from finite domains and a set of constraints between these variables. A solution to a CSP is an assignment of values to variables satisfying all constraints, and the problem amounts to finding one or all solutions. Most problems in this class are NP-complete, which means that backtracking search is an important technique in their solution.

Many search algorithms (see e.g. [2, 6–8, 11, 19]), preprocessing techniques, and constraint algorithms (see e.g. [12, 14, 16, 18, 27]) have been designed and analyzed for this class of problems (see the reviews [13, 20] for a comprehensive overview of this area). In this paper, we are mainly concerned with (network) consistency techniques, and arc consistency in particular. Consistency techniques are constraint algorithms that reduce the search space by removing, from the domains and constraints, values that cannot appear in a solution. Arc-consistency algorithms work on binary CSP and make such that the constraints are individually consistent. Arc-consistency algorithms have a long history of their own; they originate from the Waltz filtering algorithm [27] and were refined several times [12] to culminate in the optimal algorithm AC-4 of Mohr and Henderson [16]. AC-4 runs in $O(ed^2)$, where e is the number of arcs in the network and d is the size of the largest domain.

Consistency techniques have recently[1] been applied in the design of constraint logic programming (CLP) languages, more precisely in the design and implementation of CHIP [5, 24]. CHIP allows the solving of a variety of constraints over finite domains, including numerical, symbolic, and user-defined constraints. It has been applied to a variety of industrial problems and preserves the efficiency of imperative languages while shortening the development time significantly. Examples of applications include graph coloring, warehouse location, car sequencing, and cutting stock (see for instance [4, 24]). The kernel of CHIP for finite domains is an arc-consistency algorithm based on AC-3 for a set of basic binary constraints. Other (non-basic) constraints are approximated in terms of the basic constraints.

The research presented here originated as an attempt to improve further the efficiency of the kernel algorithm. This paper makes two contributions. First,

[1] Note that, Mackworth [12] mentioned as early as 1977 the potential value of consistency techniques for programming languages.

we present a new generic arc-consistency algorithm AC-5. The algorithm is generic in the sense that it is parametrized on two procedures that are specified but whose implementation is left open. It can be reduced to AC-3 and AC-4 by proper implementations of the two procedures. Moreover, we show that AC-5 can be specialized to produce an O(ed) arc-consistency algorithm for important classes of constraints: functional, anti-functional, and monotonic constraints, as well as their piecewise forms. Second, we show that the kernel of CHIP consists precisely of functional and monotonic constraints and that AC-5, in conjunction with node consistency, provides a decision procedure for the basic constraints running in time O(ed).

This paper is organized as follows. Section 2 describes the notation used in this paper and contains the basic definitions. Section 3 describes the generic arc-consistency algorithm AC-5 and specifies two abstract procedures ArcCons and LocalArcCons. Section 4 presents various representations for the domains. Sections 5–7 show how an O(ed) algorithm can be achieved for various classes of constraints by giving particular implementations of the two procedures. Section 8 introduces the concept of piecewise constraints, and Sections 9–11 extend the results for piecewise functional, anti-functional, and monotonic constraints. Section 12 shows that AC-5, in conjunction with node consistency, provides an O(ed) decision procedure for the basic constraints of CLP over finite domains. Sections 13 and 14 discuss related work and state the conclusions of this research.

2. Preliminaries

We take the following conventions. Variables are represented by the natural numbers $1, \ldots, n$. Each variable i has an associated finite domain D_i. All constraints are binary and relate two distinct variables. If i and j are variables ($i < j$), we assume, for simplicity, that there is at most one constraint relating them, denoted C_{ij}. As usual, $C_{ij}(v, w)$ denotes the boolean value obtained when variables i and j are replaced by values v and w respectively. We also denote by D the union of all domains and by d the size of the largest domain.

Arc-consistency algorithms generally work on the graph representation of the CSP. We associate a graph G to a CSP in the following way. G has a node i for each variable i. For each constraint C_{ij} relating variables i and j ($i < j$), G has two directed arcs, (i, j) and (j, i). The constraint associated to arc (i, j) is C_{ij} and the constraint associated to (j, i) is C_{ji}, which is similar to C_{ij} except that its arguments are interchanged. We denote by e the number of arcs in G. We also use $arc(G)$ and $node(G)$ to denote the set of arcs and the set of nodes of graph G.

We now reproduce the standard definitions of arc consistency for an arc and a graph.

Definition 1. Let $(i, j) \in arc(G)$. Arc (i, j) is *arc-consistent with respect to* D_i *and* D_j iff $\forall v \in D_i$, $\exists w \in D_j$: $C_{ij}(v, w)$.

Definition 2. Let $\mathscr{P} = D_1 \times \cdots \times D_n$. A graph G is *arc-consistent will respect to* $\mathscr{P}$ iff $\forall (i, j) \in arc(G)$: (i, j) is arc-consistent with respect to D_i and D_j.

The next definition is useful in specifying the outcome of an arc-consistent algorithm.

Convention 3. Let $\mathscr{P} = D_1 \times \cdots \times D_n$ and $\mathscr{P}' = D_1' \times \cdots \times D_n'$. $\mathscr{P} \sqcup \mathscr{P}'$ is defined as $(D_1 \cup D_1') \times \cdots \times (D_n \cup D_n')$, and $\mathscr{P} \sqsubseteq \mathscr{P}'$ is defined as $(D_1 \subseteq D_1')$ $\& \cdots \& (D_n \subseteq D_n')$.

Definition 4. Let $\mathscr{P} = D_1 \times \cdots \times D_n$, $\mathscr{P}' = D_1' \times \cdots \times D_n'$, and $\mathscr{P}' \sqsubseteq \mathscr{P}$. $\mathscr{P}'$ is *the largest arc-consistent domain for* G *in* $\mathscr{P}$ iff G is arc-consistent with respect to $\mathscr{P}'$ and there is no other $\mathscr{P}''$ with $\mathscr{P}' \sqsubset \mathscr{P}'' \sqsubseteq \mathscr{P}$ such that G is arc-consistent with respect to $\mathscr{P}''$.

We now show that the largest arc-consistent domain always exists and is unique.

Theorem 5 (Existence and uniqueness). *Let* $\mathscr{P} = D_1 \times \cdots \times D_n$. *The largest arc-consistent domain for* G *in* $\mathscr{P}$ *exists and is unique.*

Proof. To prove uniqueness, note that if G is arc-consistent with respect to $\mathscr{P}'$ and with respect to $\mathscr{P}''$, then G is also arc-consistent with respect to $\mathscr{P}' \sqcup \mathscr{P}''$. Hence the union of all the arc-consistent domains (included by $\sqsubseteq$ in $\mathscr{P}$) for G is also arc-consistent and is the largest arc-consistent domain for G in $\mathscr{P}$ by construction. Existence is straightforward since $\emptyset \times \cdots \times \emptyset$ is arc-consistent. □

The purpose of an arc-consistency algorithm is, given a graph G and a set $\mathscr{P}$, to compute $\mathscr{P}'$, the largest arc-consistent domain for G in $\mathscr{P}$.

3. The new arc-consistency algorithm

All algorithms for arc consistency work with a queue containing elements to reconsider. In AC-3, the queue contains arcs (i, j), while AC-4 contains pairs (i, v), where i is a node and v is a value. The novelty of AC-5 is that its queue contains elements $\langle (i, j), w \rangle$, where (i, j) is an arc and w is a value that has been removed from D_i and justifies the need to reconsider arc (i, j).

To present AC-5, we proceed in several steps. We first present the necessary operations on queues. Then we give the specification of the two abstract procedures ARCCONS and LOCALARCCONS. Finally we present the algorithm itself and prove a number of results.

3.1. Operations on queues

The operations we need are described in Fig. 1. Procedure INITQUEUE simply initializes the queue to an empty set. Function EMPTYQUEUE tests if the queue is empty. Procedure ENQUEUE(i, Δ, Q) is used whenever the set of values Δ is removed from D_i. It introduces elements of the form $\langle (k, i), v \rangle$ in the queue Q where (k, i) is an arc of the constraint graph and $v \in \Delta$. Procedure DEQUEUE dequeues one element from the queue. In all specifications, we take the convention that a parameter p subscripted with 0 (i.e. p_0) represents the value of p at call time.

All these operations on queues except Procedure ENQUEUE can be achieved in constant time. Procedure ENQUEUE can be implemented to run in O(s), where s is the number of new elements to insert in the queue. The only difficulty in fact is Procedure ENQUEUE. It requires a direct access from a variable to its arcs (which is always assumed in arc-consistency algorithms). For most algorithms, Procedure ENQUEUE can be implemented to run in O(Δ) by using a lazy distribution of v on the arcs. To achieve this result, the queue can be organized to contain elements of the form $\langle \{A_1, \ldots, A_m\}, v \rangle$, where A_k is an arc and v is a value. Procedure ENQUEUE(i, Δ, Q) adds an element $\langle \{A_1, \ldots, A_m\}, v \rangle$ to the queue, where the A_k are arcs of the form (j, i), for each $v \in \Delta$. Procedure DEQUEUE picks up an element $\langle \{A_1, \ldots, A_m\}, w \rangle$ with $m > 0$, removes an $A_k = (i, j)$ from the set, and returns i, j, and w.

---

procedure INITQUEUE(**out** Q)
Post: $Q = \{\}$.

function EMPTYQUEUE(**in** Q): Boolean
Post: EMPTYQUEUE $\Leftrightarrow$ $(Q = \{\})$.

procedure DEQUEUE(**inout** Q, **out** i, j, w)
Post: $\langle (i, j), w \rangle \in Q_0$ and $Q = Q_0 \backslash \langle (i, j), w \rangle$.

procedure ENQUEUE(**in** i, Δ, **inout** Q)
Pre: $\Delta \subseteq D_i$ and $i \in node(G)$.
Post: $Q = Q_0 \cup \{\langle (k, i), v \rangle \mid (k, i) \in arc(G)$ and $v \in \Delta\}$.

Fig. 1. The QUEUE module.

3.2. Specification of the parametric procedures

Figure 2 gives the specification of the two subproblems. Their implementations for various kinds of constraints are given in the next sections. They can also be specialized to produce AC-3 and AC-4 from AC-5.

Procedure ARCCONS(i, j, Δ) computes the set of values Δ for variable i that are not supported by D_j. Procedure LOCALARCCONS(i, j, w, Δ) is used to compute the set of values in D_i no longer supported because of the removal of value w from D_j.

Note that the specification of LOCALARCCONS gives us much freedom in the result Δ to be returned. It is sufficient to compute Δ_1 to guarantee the correctness of AC-5. However, the procedure gives us the opportunity to achieve more pruning (up to Δ_2) while still preserving the soundness of the algorithm. In the extreme case where Δ_2 is computed, the element w is thus not taken into account and LOCALARCCONS has the same result as ARCCONS.[2]

3.3. Algorithm AC-5

We are now in a position to present Algorithm AC-5. The algorithm is depicted in Fig. 3 and has two main steps. In the first step, all arcs are considered once and arc consistency is enforced on each of them. Procedure REMOVE(Δ, D) removes the set of values Δ from D. The second step applies LOCALARCCONS on each of the elements of the queue, possibly generating new elements in the queue.

3.4. Properties of AC-5

We first prove the partial correctness of AC-5. Termination, which is straightforward, is proven in the complexity results.

---

procedure ARCCONS(**in** i, j, **out** Δ)
Pre: $(i, j) \in arc(G)$, $D_i \neq \emptyset$, and $D_j \neq \emptyset$.
Post: $\Delta = \{v \in D_i \mid \forall w \in D_j: \neg C_{ij}(v, w)\}$.

procedure LOCALARCCONS(**in** i, j, w, **out** Δ)
Pre: $(i, j) \in arc(G)$, $w \notin D_j$, $D_i \neq \emptyset$, and $D_j \neq \emptyset$.
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$,
with $\Delta_1 = \{v \in D_i \mid C_{ij}(v, w)$ and $\forall w' \in D_j: \neg C_{ij}(v, w')\}$,
$\Delta_2 = \{v \in D_i \mid \forall w' \in D_j: \neg C_{ij}(v, w')\}$.

Fig. 2. Specification of the procedures.

[2] In fact, Procedure LOCALARCCONS can be made even less restrictive by replacing Δ_1 by the set $\Delta_0 = \Delta_1 \cap \{v \mid \forall w': \langle (i, j), w' \rangle \in Q \Rightarrow \neg C_{ij}(v, w')\}$. This idea simplifies some of the algorithms but is not explored in this paper.

Algorithm AC-5
Post: let $\mathcal{P}_0 = D_{1_0} \times \cdots \times D_{n_0}$,
$\mathcal{P} = D_1 \times \cdots \times D_n$
G is maximally arc-consistent with respect to $\mathcal{P}$ in $\mathcal{P}_0$.
begin AC-5
1 INITQUEUE(Q)
2 **for each** $(i, j) \in arc(G)$ **do**
3 **begin**
4 ARCCONS(i, j, Δ);
5 ENQUEUE(i, Δ, Q);
6 REMOVE(Δ, D_i)
7 **end**;
8 **while not** EMPTYQUEUE(Q) **do**
9 **begin**
10 DEQUEUE(Q, i, j, w);
11 LOCALARCCONS(i, j, w, Δ);
12 ENQUEUE(i, Δ, Q);
13 REMOVE(Δ, D_i)
14 **end**
end AC-5

Fig. 3. The arc-consistency algorithm AC-5.

Lemma 6. *Let $\mathcal{P}^* = D_1^* \times \cdots \times D_n^*$ be the largest arc-consistent domain for G in $\mathcal{P}_0$. The invariant $\mathcal{P}^* \sqsubseteq \mathcal{P}$ is preserved in AC-5 at lines 2 and 8.*

Proof. The invariant holds for the first execution of line 2, since $D_i = D_{i_0}$ and $D_i^* \subseteq D_{i_0}$. Execution of line 4 preserves the invariant because $v \in \Delta \Rightarrow v \notin D_i^*$, since $\mathcal{P}^* \sqsubseteq \mathcal{P}$ and $\mathcal{P}^*$ is arc-consistent. It follows that $D_i^* \subseteq D_i \backslash \Delta$ and lines 5 and 6 also preserve the invariant. The proof for the invariant in line 8 is similar. □

Theorem 7 (Partial correctness). *Algorithm AC-5 is partially correct.*

Proof. We first show that G is arc-consistent with respect to $\mathcal{P}$ when AC-5 terminates. If we assume the contrary, there must exist $(i, j) \in arc(G)$ and $v \in D_i$ such that $\forall w \in D_j$: $\neg C_{ij}(v, w)$. The value v must then be supported by some elements of D_{j_0}, otherwise it would have been removed from D_i at line 6. Let $w_1, \ldots, w_m$ $(m > 0)$ be all the elements of D_{j_0} supporting v. The values w_k $(1 \leq k \leq m)$ are thus removed from D_{j_0} during the execution, and elements of the form $\langle (j, i), w_k \rangle$ are inserted in the queue. Since AC-5 terminates, LOCALARCCONS(j, i, w_l, Δ) in line 11 is executed from some l $(1 \leq l \leq m)$ with

procedure INITQUEUE(**out** Q)
Post: $\forall(k, i) \in arc(G)$:
$Status[(k, i), v] =$ `present` if $v \in D_i$,
$=$ `rejected` if $v \notin D_i$.

function EMPTYQUEUE(**in** Q)
Post: $\forall(k, i) \in arc(G), \forall v$:
$Status[(k, i), v] \neq$ `suspended`.

procedure DEQUEUE(**inout** Q, **out** i, j, w)
Post: $Status[(i, j), w] =$ `rejected`.

procedure ENQUEUE(**in** i, Δ, **inout** Q)
Pre: $\forall(k, i) \in arc(G), \forall v \in \Delta$:
$Status[(k, i), v] =$ `present`.
Post: $\forall(k, i) \in arc(G), \forall v \in \Delta$:
$Status[(k, i), v] =$ `suspended`.

Fig. 4. The QUEUE module on structure *Status*.

$w_k \notin D_j$ for all k $(1 \leq k \leq m)$. By definition of LOCALARCCONS, $v \in \Delta$ holds and line 13 removes v from D_i, resulting in a contradiction.

Now, since $\mathscr{P}^* \sqsubseteq \mathscr{P}$ by Lemma 6, where $\mathscr{P}^*$ is the largest arc-consistent domain for G in $\mathscr{P}_0$, it follows that $\mathscr{P} = \mathscr{P}^*$. This proves the partial correctness of AC-5. □

We now turn to the complexity results. To simplify the presentation, we introduce a new data structure *Status* which is a two-dimensional array, the first dimension being on arcs and the second on values. We also give the effect of the procedures manipulating the queue on *Status* in Fig. 4. Note that the actual implementation does not need to perform these operations; they are just presented here merely to ease the presentation and simplify the theorem.

Algorithm AC-5 preserves the following invariant on lines 2 and 8 for *Status*:

$$\begin{aligned} Status[(k, i), v] &= \texttt{present} && \text{iff} \quad v \in D_i, \\ &= \texttt{suspended} && \text{iff} \quad v \notin D_i \ \&\ \langle (k, i), v \rangle \in Q, \\ &= \texttt{rejected} && \text{iff} \quad v \notin D_i \ \&\ \langle (k, i), v \rangle \notin Q. \end{aligned}$$

We are now in a position to prove the following theorem.

Theorem 8. *Algorithm AC-5 has the following properties*:

(1) *The invariant on data structure Status holds on lines 2 and 8.*

(2) *AC-5 enqueues and dequeues at most* $O(ed)$ *elements, and hence the size of the queue is at most* $O(ed)$.

(3) *AC-5 always terminates.*

(4) *If* $s_1, \ldots, s_p$ *are the numbers of new elements in the queue on each iteration at lines 5 and 12, then* $s_1 + \cdots + s_p \leq O(ed)$.

Proof. Property (1) holds initially. Assuming that it holds in line 2, it also holds after an iteration of lines 4–6. Line 5 makes sure that $\langle (j, i), v) \rangle$ is `suspended` for all $v \in \Delta$ and puts them on the queue, while line 6 removes Δ from D_i. So the invariant holds at the first execution of line 8. Execution of lines 10–13 preserves the invariant, lines 10 and 11 maintain it on their own, and lines 12 and 13 respectively make sure that $\langle (j, i), v \rangle$ is `suspended` for all $v \in \Delta$ and remove Δ from D_i.

Property (2) holds because each element of *Status* is allowed to make only two transitions: one from `present` to `suspended` through Procedure ENQUEUE and one from `suspended` to `rejected` through Procedure DEQUEUE. Hence there can only be $O(ed)$ dequeues and enqueues.

Properties (3) and (4) are direct consequences of property (2) and the preconditions of ENQUEUE on the data structure *Status*. □

The space complexity of AC-5 depends on the maximal size of Q and on the size of the domains of the variables. The above theorem can be used to deduce the overall complexity of AC-5 from the complexity of Procedures ARCCONS and LOCALARCCONS.

Theorem 9.

(1) *If the time complexity of* ARCCONS *is* $O(d^2)$ *and the time complexity of* LOCALARCCONS *is* $O(d)$, *then the time complexity of AC-5 is* $O(ed^2)$.

(2) *If the time complexity of* ARCCONS *is* $O(d)$ *and the time complexity of* LOCALARCCONS(i, j, w, Δ) *is* $O(\Delta)$,[3] *then the time complexity of AC-5 is* $O(ed)$.

AC-3 is a particular case of AC-5 where the value w is never used in the implementation of Procedure LOCALARCCONS[4] (i.e. LOCALARCCONS is implemented by ARCCONS). In this case, LOCALARCCONS and ARCCONS are $O(d^2)$ and AC-5 is $O(ed^3)$. The space complexity is $O(e + nd)$, since the size of the queue can be reduced to $O(e)$.

AC-4 is also a particular case of AC-5 where the implementation of Procedure LOCALARCCONS does not use node i, but maintains a data structure of size $O(ed^2)$. In this case, ARCCONS initializes the data structure and is $O(d^2)$, and LOCALARCCONS is $O(d)$. The resulting algorithm is $O(ed^2)$.

[3] $O(\Delta)$ really means $O(\max(1, |\Delta|))$, since it should be $O(1)$ when Δ is empty.

[4] Strictly speaking, in AC-3, arc(i, j) is not enqueued when arc(j, i) is made consistent. This optimization could be added in AC-5 by adding j as an argument to ENQUEUE and adding the constraint $k \neq j$ to its definition.

Since $O(ed^2)$ is the optimal time complexity, there is no way to reduce the complexity other than considering particular classes of constraints, allowing us to implement, in particular, Procedure ArcCons in $O(d)$. Note also that an arc-consistency algorithm in $O(ed)$ is optimal for a subclass of constraints, since it is reasonable to assume that we need to check each value in each domain at least once. In the following sections, we characterize classes of constraints that guarantee that Procedure ArcCons is $O(d)$ and Procedure LocalArcCons is linearly related to the size of its output set Δ, hence resulting in an AC-5 algorithm for these classes running in time $O(ed)$ and space $O(ed + nd)$.

4. Representation of domains

Particular implementations of ArcCons and LocalArcCons perform operations on the domains depicted in Fig. 5. As the reader will notice, the operations we define on the domains are more sophisticated than those usually required by arc-consistency algorithms. In particular, they assume a total

---

function Size(**in** D): Integer
Post: Size $= |D|$.

procedure RemoveElem(**in** v, **inout** D)
Post: $D = D_0 \backslash \{v\}$.

function Member(**in** v, D): Boolean
Post: Member $\Leftrightarrow$ $(v \in D)$.

function Min(**in** D): Value
Post: Min $= \min\{v \in D\}$.

function Max(**in** D): Value
Post: Max $= \max\{v \in D\}$.

function Succ(**in** v, D): Value
Post: Succ $= \min\{v' \in D \mid v' > v\}$, if $\exists v' \in D: v' > v$,
$= -\infty$, otherwise.

function Pred(**in** v, D): Value
Post: Pred $= \max\{v' \in D \mid v' < v\}$, if $\exists v' \in D: v' < v$,
$= -\infty$, otherwise.

Fig. 5. The Domain module.

ordering on the domain D for reasons that will become clear later.[5] The additional sophistication is necessary to achieve the bound O(ed) for monotonic constraints.

The primitive operations on domains are assumed to take constant time. We present here two data structures that enable to achieve this result.

The first data structure assumes a domain of consecutive integer values and is depicted in Fig. 6. The field *size* gives the size of the domain, the fields *min* and *max* are used to pick up the minimum and maximum values, the field *element* to test if a value is in the domain, and the two fields *pred* and *succ* to access in constant time the successor or predecessor of a value in the domain. The operation REMOVEELEMENT must update all fields to preserve the semantics. This can be done in constant time.

When the domain is sparse, the data structure depicted in Fig. 7 can be used. It reasons about indices instead of values and uses a hash table to test membership in the domain. Although the time complexity of membership is theoretically not O(1), under reasonable assumptions, the expected time to search for an element is O(1) [1].

For ease of presentation, we assume in the rest of the paper that AC-5 stops as soon as a domain becomes empty.

Let $S = \{b, \ldots, B\}$

$D_i = \{v_1, \ldots, v_m\} \subseteq S$ with $v_k < v_{k+1}$ and $m > 0$.

Syntax

$D_i.size$: integer

$D_i.min$: integer $\in S$

$D_i.max$: integer $\in S$

$D_i.element$: array $[b..B]$ of booleans

$D_i.succ$: array $[b..B]$ of integers $\in S$

$D_i.pred$: array $[b..B]$ of integers $\in S$

Semantics

$D_i.size = m$

$D_i.min = v_1$

$D_i.max = v_m$

$D_i.element[v]$ iff $v \in D_i$

$D_i.succ[v_k] = v_{k+1}$ $(1 \leq k < m)$

$D_i.succ[v_m] = +\infty$

$D_i.pred[v_{k+1}] = v_k$ $(1 \leq k < m)$

$D_i.pred[v_1] = -\infty$

Fig. 6. DOMAIN data structure: consecutive values.

[5] Note that if D is made up of several unconnected domains with distinct orderings, it is always possible to transform the underlying partial ordering into a total ordering.

Let $S = \{e_1, \ldots, e_a\}$ with $e_k < e_{k+1}$
$D_i = \{e_{v_1}, \ldots, e_{v_m}\} \subseteq S$ with $v_k < v_{k+1}$ and $m > 0$

Syntax

$D_i.size$: integer
$D_i.min$: integer $\in \{1, \ldots, a\}$
$D_i.max$: inter $\in \{1, \ldots, a\}$
$D_i.element$:
set of couples (e, v) with $e \in S$ and $v \in \{0, \ldots, a\}$,
organized as a hash table on key e.
$D_i.value$: array $[1..a]$ of elements $\in S$
$D_i.succ$: array $[1..a]$ of integers $\in \{1, \ldots, a\}$
$D_i.pred$: array $[1..a]$ of integers $\in \{1, \ldots, a\}$

Semantics

$D_i.size = m$
$D_i.min = v_1$
$D_i.max = v_m$
$D_i.element(e) = v$ (with $e = e_v$), if $e \in D_i$
$= 0$, otherwise
$D_i.value[v] = e_v$
$D_i.succ[v_k] = v_{k+1}$ $(1 \leq k < m)$
$D_i.succ[v_m] = +\infty$
$D_i.pred[v_{k+1}] = v_k$ $(1 \leq k < m)$
$D_i.pred[v_1] = -\infty$

Fig. 7. DOMAIN data stucture: sparse values.

5. Functional constraints

Definition 10. A constraint C is *functional* with respect to a domain D iff for all $v \in D$ (respectively $w \in D$) there exists at most one $w \in D$ (respectively $v \in D$) such that $C(v, w)$.

Note that the above definition is parametrized on a domain D. Some constraints might not be functional in general but become functional when restricted to a domain of values. An example of a functional constraint is $x = y + 5$.

Convention 11. If C_{ij} is a functional constraint, we denote by $f_{ij}(v)$ (respectively $f_{ji}(w)$) the value w (respectively v) such that $C_{ij}(v, w)$. If such a value does not exist, the function denotes a value outside the domain for which the constraint holds.

```
procedure ArcCons(in i, j, out Δ)
   begin
1     Δ := ∅;
2     for each v ∈ D_i do
3        if f_ij(v) ∉ D_j then
4           Δ := Δ ∪ {v}
   end
```

Fig. 8. ArcCons for functional constraints.

The results presented in the paper assume that it takes constant time to compute the functions f_{ij} and f_{ji} in the same way as arc-consistency algorithms assume that $C(v, w)$ can be computed in constant time.

We can now present procedures ArcCons and LocalArcCons for functional constraints, as depicted in Figs. 8 and 9. It is clear that the procedures fulfill their specifications. Only one value per arc needs to be checked in Procedure ArcCons since the constraint is functional. Procedure LocalArcCons computes the set Δ_1 in this case and only one value needs to be checked. Procedures ArcCons and LocalArcCons are respectively $O(d)$ and $O(1)$ for functional constraints. Hence we have an optimal algorithm.

Theorem 12. *Algorithm AC-5 is* $O(ed)$ *for functional constraints with respect to* D.

Note that functional constraints add no requirement for the basic operations on the domains compared to traditional algorithms.

6. Anti-functional constraints

When the negation of a constraint is functional (for instance, the inequality relation $x \neq y$), an optimal algorithm can also be achieved.

```
procedure LocalArcCons(in i, j, w, out Δ)
   begin
1     if f_ji(w) ∈ D_i then
2        Δ := {f_ji(w)}
3     else
4        Δ := ∅
   end
```

Fig. 9. LocalArcCons for functional constraints.

```
procedure ArcCons (in i, j, out Δ)
  begin
1   s := Size(D_j);
2   w_1 := Min(D_j);
3   if s = 1 then
4     Δ := {f_ji(w_1)} ∩ D_i
5   else
6     Δ := ∅
7   end
  end
```

Fig. 10. Procedure ArcCons for anti-functional constraints.

Definition 13. A constraint C_{ij} is *anti-functional* with respect to a domain D iff $\neg C_{ij}$ is functional with respect to D.

With an anti-functional constraint, for each value in the domain there is thus at most one value for which the constraint does not hold. Procedures ArcCons and LocalArcCons are shown in Figs. 10 and 11. We use the same convention as for functional constraints.

Instead of considering each element of D_i, which would yield a complexity $O(d)$, the result of ArcCons is here achieved by considering the size of D_j. It is clear that ArcCons fulfills its specification: for $D_j = \{w\}$, the resulting set should contain $f_{ij}(w)$ only if it is an element of D_i. The complexity of ArcCons is O(1). This allows the implementation of LocalArcCons through ArcCons, leading to the same O(1). In this case, the value w is not considered and LocalArcCons computes the set Δ_2 of its specification.[6]

Theorem 14. *Algorithm AC-5 is $O(ed)$ for anti-functional constraints with respect to D.*

```
procedure LocalArcCons(in i, j, w, out Δ)
  begin
1   ArcCons(i, j, Δ)
  end
```

Fig. 11. Procedure LocalArcCons for anti-functional constraints.

[6] The set Δ_1 can also be computed in O(1) since one can show that $\Delta_1 = \Delta_2 \backslash \{f_{ij}(w)\}$.

7. Monotonic constraints

We now consider another class of constraints: monotonic constraints, for example $x \leq y - 3$. This class of constraints requires a total ordering $<$ on D, as mentioned previously. Moreover, we assume that, for any constraint C and element $v \in D$, there exist elements w_1 and w_2 (not necessarily in D) such that $C(v, w_1)$ and $C(w_2, v)$ hold. This last requirement is used to simplify the algorithms but it is not restrictive in nature.

Definition 15. A constraint C is *monotonic* with respect to a domain D iff there exists a total ordering on D such that, for all values v and w in D, $C(v, w)$ holds implies $C(v', w')$ holds for all values v' and w' in D such that $v' \leq v$ and $w' \geq w$.

Convention 16. Since AC-5 is working with arcs, we associate with each arc (i, j) three functions f_{ij}, $last_{ij}$, and $next_{ij}$ and a relation $>_{ij}$. Given a monotonic constraint C_{ij}, the functions and relation for arc (i, j) are

$$f_{ij}(w) = \max\{v \mid C_{ij}(v, w)\}, \qquad last_{ij} = \text{MAX},$$
$$next_{ij} = \text{PRED}, \qquad >_{ij} = >$$

while those for arc (j, i) are

$$f_{ji}(v) = \min\{w \mid C_{ij}(v, w)\}, \qquad last_{ji} = \text{MIN},$$
$$next_{ji} = \text{SUCC}, \qquad >_{ji} = <.$$

Moreover, since Procedures ARCCONS and LOCALARCCONS only use f_{ij}, $last_{ij}$, $next_{ij}$, and $>_{ij}$ for arc (i, j), we omit the subscripts in the presentation of the algorithms. These functions are assumed to take constant time to evaluate.

We are now in a position to describe the implementation of Procedures ARCCONS and LOCALARCCONS for monotonic constraints. They are depicted in Figs. 12 and 13.

Lemma 17. *Procedures* ARCCONS *and* LOCALARCCONS *fulfill their specifications.*

Proof. Procedures ARCCONS and LOCALARCCONS compute the set $\Delta = \{v \in D_i \mid v > f(last(D_j))\}$. By monotonicity of the constraint, $\Delta \subseteq \Delta_2$ with $\Delta_2 = \{v \in D_i \mid \forall w' \in D_j\colon \neg C_{ij}(v, w')\}$, and $\Delta_2 \cap \{v \in D_i \mid v \leq f(last(D_j))\} = \emptyset$. Hence $\Delta = \Delta_2$ and both postconditions are satisfied. □

Procedures ARCCONS and LOCALARCCONS have as many iterations in lines 5 and 6 as there are elements in the resulting set Δ. Hence it follows that we have an optimal algorithm.

```
procedure ArcCons(in i, j, out Δ)
   begin
1     Δ := ∅
2     v := last(D_i);
3     while v > f(last(D_j)) do
4       begin
5         Δ := Δ ∪ {v};
6         v := next(v, D_i)
7     end
   end
```

Fig. 12. ArcCons for monotonic constraints.

Theorem 18. *Procedure AC-5 is* O(*ed*) *for monotonic constraints with respect to D.*

It is also clear that AC-5 can be applied at the same time to (anti-)functional and monotonic constraints with the same complexity.

Monotonic constraints revisited

Let us reconsider the ArcCons procedure for monotonic constraints. We first show that the Succ and Pred functions can always be applied on the initial domains (denoted D_i^{init}), thus eliminating the need to update part of the data structure. The revised procedure ArcCons is depicted in Fig. 14. The only difference lies in lines 5 and 6, and thus obviously has no influence on the correctness of ArcCons.

Procedure LocalArcCons could use ArcCons, but a revised version is presented in Fig. 15. The correctness of LocalArcCons is a consequence of the preceding version, computing the set Δ_2 of its specification, and the fact that when $w \leq last(D_j)$, then Δ_1 is empty by the monotonicity of C_{ij}. It is possible to compute Δ_1,[7] but this would prevent the reduction of domains as early as possible.

```
procedure LocalArcCons(in i, j, in w, out Δ)
   begin
1     ArcCons(i, j, Δ)
   end
```

Fig. 13. LocalArcCons for monotonic constraints.

[7] In line 4 in Fig. 15, replace $f(last(D_j))$ by $f(w)$.

```
procedure ArcCons(in i, j, out Δ)
   begin
1    Δ := ∅;
2    v := last(D_i);
3    while v > f(last(D_j)) do
4      begin
5        if v ∈ D_i then
             Δ := Δ ∪ {v};
6        v := next(v, D_i^init)
7    end
   end
```

Fig. 14. Revised Procedure ArcCons for monotonic constraints.

Theorem 19. *With the revised implementation depicted in Figs. 14 and 15, Procedure AC-5 is* O(*ed*) *for monotonic constraints with respect to D.*

Proof. This proof requires the use of amortized complexity [22] to show that LocalArcCons is O(d) amortized. The number of iterations for a call to the revised version of LocalArcCons is not O(d) in the worst case, since some elements may have been removed from the domain. However, we can associate, to each arc (i, j), d credits that are used each time a test in line 5 (ArcCons) or in line 7 (LocalArcCons) is executed for arc (i, j) and no element is inserted. The total number of credits is thus O(ed). To prove the amortized O(d) complexity, we show that a test in line 5 (ArcCons) or in line

```
procedure LocalArcCons(in i, j, in w, out Δ)
   begin
1    Δ := ∅;
2    if w > last(D_j) then
3      begin
4        v := last(D_i);
5        while v > f(last(D_j)) do
6          begin
7            if v ∈ D_i then
                 Δ := Δ ∪ {v};
8            v := next(v, D_i^init)
9          end
10     end
   end
```

Fig. 15. Revised Procedure LocalArcCons for monotonic constraints.

7 (LocalArcCons) is done at most once per value in the domain. Suppose that such a test is done on some v'. Then, after the execution of the following Remove, we have $v' > last(D_i)$, and this value is thus never considered again, since in each execution of ArcCons and LocalArcCons, the first execution of the test always succeeds. Hence, it follows from the number of credits and the complexity of the first algorithm that we still have an optimal AC-5 algorithm. □

8. Piecewise constraints

The preceding sections are generalized to the case when the domain can be partitioned into groups such that elements of a group behave similarly with respect to a given constraint.

Convention 20. Let S and P be sets, and C be a constraint. $C(S, P)$ denotes $\forall v \in S, \forall w \in P$: $C(v, w)$, and $\neg C(S, P)$ denotes $\forall v \in S, \forall w \in P$: $\neg C(v, w)$. We also use $C(S, w)$ for $C(S, \{w\})$.

Definition 21. The partitions $\mathcal{S} = \{S_0, \ldots, S_n\}$ of D_i and $\mathcal{P} = \{P_0, \ldots, P_m\}$ of D_j are a *piecewise decomposition* of D_i and D_j with respect to C iff for all $S_k \in \mathcal{S}$ and $P_{k'} \in \mathcal{P}$: $C(S_k, P_{k'})$ or $\neg C(S_k, P_{k'})$ holds.

Representation of piecewise constraints

Before presenting the implementation of ArcCons and LocalArcCons for constraints having some particular piecewise decomposition, we show in Fig. 16 the operations on piecewise decompositions. For ease of implementation, we assume that elements in the groups of a piecewise decomposition are never removed during the execution. The piecewise decomposition of D_i and D_j with respect to C_{ij} is denoted $\mathcal{S}_{ij} = \{S_0^{ij}, \ldots, S_n^{ij}\}$ and $\mathcal{S}_{ji} = \{S_0^{ji}, \ldots, S_m^{ji}\}$. We also introduce a new data structure *Status-pd* which is a two-dimensional array, the first dimension being on arcs (associated with a piecewise decomposition) and the second on group numbers. Its semantics is the following:

$$S_k^{ij} \cap D_i \neq \emptyset \Rightarrow \textit{Status-pd}[(i, j), k] = \texttt{false}.$$

Thus, *Status-pd* must be `false` when the corresponding group is not empty.

The primitive operations on a piecewise decomposition are assumed to take constant time, except that the complexity of Extend is assumed to be $O(s)$, where s is the size of S_k^{ij}.

A simple data structure that enables us to achieve these results is given in Fig. 17. Its space complexity is $O(d)$ per piecewise decomposition. This data

function NBGROUP(**in** i, j): Integer
Post: NBGROUP $= |\mathcal{S}_{ij}| - 1$.

function SIZEOFGROUP(**in** i, j, k): Integer
Pre: $0 \leq k \leq$ NBGROUP(i, j).
Post: SIZEOFGROUP $= |S_k^{ij} \cap D_i|$.

function EMPTYGROUP(**in** i, j, k): Boolean
Pre: $0 \leq k \leq$ NBGROUP(i, j).
Post: EMPTYGROUP $\Leftrightarrow$ $S_k^{ij} \cap D_i = \emptyset$.

procedure EXTEND(**in** i, j, k, **inout** Δ)
Pre: $0 \leq k \leq$ NBGROUP(i, j).
Post: $\Delta = \Delta_0 \cup (S_k^{ij} \cup D_i)$,
Status-pd$[(i, j), k]$ = `true`.

function GROUPOF(**in** i, j, v): Integer
Pre: $v \in D_i^{\text{init}}$,
Post: GROUPOF $= k$ such that $v \in S_k^{ij}$.

function FIRSTGROUP(**in** i, j): Integer
Post: FIRSTGROUP $= \min\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}$.

function LASTGROUP(**in** i, j): Integer
Post: LASTGROUP $= \max\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}$.

function SIZE(**in** i, j): Integer
Post: SIZE $= |\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}|$.

Fig. 16. The PIECEWISE DECOMPOSITION module.

structure cannot be updated by the REMOVEELEM primitive in constant time since an element in a domain can belong to different groups in different piecewise decompositions. The update can easily be performed by the ENQUEUE primitive, however, without affecting its complexity.

It is not difficult to initialize the data structure in $O(d)$ under the realistic assumption that it takes $O(s)$ to find the s elements in D_j (respectively D_i) supporting a value v (respectively w) in D_i (respectively D_j). In addition, the construction of the data structure assigns a group number to each value, so that the GROUPOF operation trivially takes constant time. In the following, we assume that the data structure has already been built.

9. Piecewise functional constraints

Intuitively, a piecewise functional constraint C_{ij} is a constraint whose do-

Let $\mathcal{S}_{ij} = \{S_0^{ij}, \ldots, S_n^{ij}\}$ with $n \geq 0$

Syntax

$S_{ij}.group$: array $[1..n]$ of sets
$S_{ij}.nbgroup$: integer
$S_{ij}.size$: integer
$S_{ij}.sizegroup$: array $[1..n]$ of integers
$S_{ij}.first$: integer
$S_{ij}.last$: integer

Semantics

$S_{ij}.group[k] = S_k^{ij}$
$S_{ij}.nbgroup = n$
$S_{ij}.size = |\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}|$
$S_{ij}.sizegroup[k] = |S_k^{ij} \cap D_i|$
$S_{ij}.first = \min\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}$
$S_{ij}.last = \max\{k \mid S_k^{ij} \cap D_i \neq \emptyset\}$

Fig. 17. PIECEWISE DECOMPOSITION data structure.

mains can be decomposed into groups such that each group of D_i (respectively D_j) is supported by at most one group of D_j (respectively D_i).

Definition 22. A constraint C_{ij} is *piecewise functional* with respect to domains D_i and D_j iff there exists a piecewise decomposition $\mathcal{S} = \{S_0, \ldots, S_n\}$ and $\mathcal{P} = \{P_0, \ldots, P_m\}$ of D_i and D_j with respect to C_{ij} such that for all $S_k \in \mathcal{S}$ (respectively $P_{k'} \in \mathcal{P}$), there exists at most one $P_{k'} \in \mathcal{P}$ (respectively $S_k \in \mathcal{S}$) such that $C_{ij}(S_k, P_{k'})$.

Examples of functional piecewise constraints are the modulo ($x = y \bmod z$) and integer division ($x = y \operatorname{div} z$) constraints. The `element` constraint of the CHIP programming language [24] is a piecewise constraint as well. Finally, note that functional constraints are a subclass of piecewise constraints, in which the size of each group in the partition is exactly one.

Obviously, in a piecewise functional constraint C_{ij}, if all the unsupported elements of D_i (respectively D_j) are in the same group (e.g. S_0 and P_0), then the piecewise decompositions $\mathcal{S} = \{S_0, \ldots, S_n\}$ and $\mathcal{P} = \{P_0, \ldots, P_n\}$ have the same number of groups and the groups can be renumbered such that the following hold:

(PF1) $\neg C_{ij}(S_0, D_j)$ and $\neg C_{ij}(D_i, P_0)$;
(PF2) $C_{ij}(S_k, P_k)$, $1 \leq k \leq n$;
(PF3) $\neg C_{ij}(S_k, P_{k'})$, $1 \leq k, k' \leq n$ and $k \neq k'$.

The implementation of ARCCONS and LOCALARCCONS for piecewise functional constraints assumes a piecewise decomposition that satisfies (PF1)–

(PF3). The following property states necessary and sufficient conditions for a piecewise functional constraint.

Property 23. *A constraint C_{ij} is piecewise functional with respect to D_i and D_j iff there exists a partition $\mathcal{S} = \{S_0, \ldots, S_n\}$ of D_i such that*

(1) $C_{ij}(S_k, w)$ or $\neg C_{ij}(S_k, w)$ *for all* $w \in D_j$ *and* $0 \leq k \leq n$;

(2) $C_{ij}(S_k, w) \Rightarrow \neg C_{ij}(S_{k'}, w)$ *for all* $w \in D_j$, $0 \leq k, k' \leq n$, *and* $k \neq k'$.

Proof. The "only if" part is straightforward. For the "if" part, let us assume that there is some unsupported element in D_i and in D_j and that all the unsupported elements in D_i are in S_0 (otherwise groups can be merged and renumbered without affecting conditions (1) and (2)). We construct $\mathcal{P} = \{P_0, \ldots, P_n\}$ in the following way:

$$P_k = \{w \in D_j \mid \exists v \in S_k, C_{ij}(v, w)\}, \quad 1 \leq k \leq n,$$

$$P_0 = D_j \setminus \bigcup_{1 \leq l \leq n} P_l.$$

It is sufficient to prove that $\mathcal{P}$ is a partition and that $\mathcal{S}$ and $\mathcal{P}$ satisfy (PF1)–(PF3).

We first prove that $\mathcal{P}$ is a partition.

(A) $P_k \cap P_{k'} = \emptyset$ $(k \neq k')$. This holds for $k = 0$ or $k' = 0$. For $k \neq 0 \neq k'$, let $w \in P_k$. By definition of P_k, we have $\exists v \in S_k$: $C_{ij}(v, w)$. Hence by (1), $C_{ij}(s_k, w)$. By (2) we have $\neg C_{ij}(S_{k'}, w)$, that is $\forall v' \in S_{k'}$: $\neg C_{ij}(v', w)$. Hence $w \notin P_{k'}$.

(B) Suppose that $P_k = \emptyset$ $(k > 0)$. Then $S_k = \emptyset$ (impossible since $\mathcal{S}$ is a partition), or S_k contains unsupported elements (impossible by hypothesis). Hence $P_k \neq \emptyset$.

Now we prove that $\mathcal{S}$ and $\mathcal{P}$ satisfy (PF1)–(PF3).

(PF1) holds by definition of S_0 and P_0.

(PF2): Let $w \in P_k$. By definition of P_k, $\exists v' \in S_k$ such that $C_{ij}(v', w)$. By (1), $C_{ij}(S_k, w)$, that is $\forall v \in S_k$: $C_{ij}(v, w)$. Hence $C_{ij}(S_k, P_k)$.

(PF3): Let $w \in P_k$. Since $P_k \cap P_{k'} = \emptyset$ $(k \neq k')$, $w \notin P_{k'}$. By definition of $P_{k'}$, we have $\forall v' \in S_{k'}$: $\neg C_{ij}(v, w)$. Hence $\neg C_{ij}(S_k, P_{k'})$. □

The procedures ArcCons and LocalArcCons for piecewise functional constraints are given in Figs. 19 and 20. Line 2 handles the group S_0^{ij} containing all the unsupported elements of the initial domain D_i. The procedures use the boolean function Unsupported specified in Fig. 18. The correctness of these procedures is an immediate consequence of the correctness of procedures for functional constraints. One can also easily see that the semantics of *Status-pd* is an invariant at lines 2 and 8 in AC-5, assuming it holds initially.

```
function UNSUPPORTED(in i, j, k): Boolean
Pre: 0 ≤ k ≤ NBGROUP(i, j).
Post: UNSUPPORTED ⇔
      EMPTYGROUP(j, i, k) ∧ ¬Status-pd[(i, j), k]
```

Fig. 18. The UNSUPPORTED function.

```
procedure ARCCONS(in i, j, out Δ)
  begin
1   Δ := ∅;
2   EXTEND(i, j, 0, Δ);
3   for k := 1 to NBGROUP(i, j) do
4     if UNSUPPORTED(i, j, k) then
5       EXTEND(i, j, k, Δ)
  end
```

Fig. 19. ARCCONS for piecewise functional constraints.

The time complexity is analyzed globally within AC-5. If the complexity of all the executions of ARCCONS and LOCALARCCONS for a given arc (i, j) is bounded by $O(d)$, then AC-5 is $O(ed)$. The complexity of execution of ARCCONS and LOCALARCCONS depends mainly on the number of executions of the EXTEND procedure. For an arc (i, j), by the specification of UNSUPPORTED and EXTEND (on *Status-pd*), at most one EXTEND operation is made per group, and hence the complexity is bounded by $O(d)$. If we use amortized complexity as in the case of monotonic constraints, it follows that we have an optimal algorithm.

Theorem 24. *Procedure AC-5 is* $O(ed)$ *for piecewise functional constraints.*

```
procedure LOCALARCCONS(in i, j, w, out Δ)
  begin
1   Δ := ∅;
2   k := GROUPOF(j, i, w);
4   if UNSUPPORTED(i, j, k) then
5     EXTEND(i, j, k, Δ)
  end
```

Fig. 20. LOCALARCCONS for piecewise functional constraints.

10. Piecewise anti-functional constraints

We now turn to piecewise anti-functional constraints such as $x \neq y \bmod 3$. A piecewise anti-functional constraint is a constraint whose domains D_i and D_j can be decomposed into groups such that each group of D_i (respectively D_j) is not supported by at most one group of D_j (respectively D_i).

Definition 25. A constraint C_{ij} is *anti-functional* with respect to D_i and D_j iff $\neg C_{ij}$ is piecewise functional with respect to D_i and D_j.

With the same notations as in the preceding section, procedures ArcCons and LocalArcCons for anti-functional constraints can easily be extended in the piecewise framework (see Figs. 21 and 22). Note the test for $k \neq 0$, since group 0 supports all groups. By a complexity analysis similar to that of the preceding section, one can show that in AC-5 there will be at most one execution of Extend per group. Hence the following result.

Theorem 26. *Algorithm AC-5 is* O(*ed*) *for piecewise anti-functional constraints.*

```
procedure ArcCons(in i, j, out Δ)
   begin
1     Δ := ∅;
2     s := Size(j, i):
3     k := FirstGroup(j, i);
4     if s = 1 and k ≠ 0
                 and not EmptyGroup(i, j, k) then
5       Extend(i, j, k, Δ)
   end
```

Fig. 21. Procedure ArcCons for piecewise anti-functional constraints.

```
procedure LocalArcCons(in i, j, w, out Δ)
   begin
1     ArcCons(i, j, Δ)
   end
```

Fig. 22. Procedure LocalArcCons for piecewise anti-functional constraints.

11. Piecewise monotonic constraints

Monotonic constraints are finally generalized to piecewise monotonic constraints, for example $x \leq y \text{ div } 5$.

Definition 27. A constraint C_{ij} is *piecewise monotonic* with respect to D_i and D_j iff there exists a piecewise decomposition $\mathcal{S} = \{S_0, \ldots, S_n\}$ and $\mathcal{P} = \{P_0, \ldots, P_m\}$ of D_i and D_j with respect to C_{ij} such that

$$C_{ij}(S_k, P_l) \Rightarrow C_{ij}(S_{k'}, P_{l'})$$

for $0 \leq k' \leq k \leq n$ and $0 \leq l \leq l' \leq m$.

Convention 28. As for monotonic constraints, we associate to each arc (i, j) three functions f_{ij}, $last_{ij}$, and $next_{ij}$ and a relation $>_{ij}$. Given a piecewise monotonic constraint C_{ij}, the functions and relation for arc (i, j) are:

$$f_{ij}(k) = \max\{\{-1\} \cup \{k' \mid C_{ij}(S_k^{ij}, S_{k'}^{ji})\}\},$$
$$last_{ij}(a, b) = \text{LastGroup}(a, b),$$
$$next_{ij}(k) = k - 1, \qquad >_{ij} = >,$$

while those for arc (j, i) are

$$f_{ji}(k) = \min\{\{\text{NbGroup}(j, i) + 1\} \cup \{k' \mid C_{ij}(S_{k'}^{ij}, S_k^{ji})\}\},$$
$$last_{ji}(a, b) = \text{FirstGroup}(a, b),$$
$$next_{ji}(k) = k + 1, \qquad >_{ji} = <.$$

```
procedure ArcCons(in i, j, out Δ)
  begin
1   Δ := ∅;
2   k := last(i, j);
3   while k > f(last(j, i)) do
4     begin
5       if not EmptyGroup(i, j, k) then
          Extend(i, j, k, Δ);
6       k := next(k)
7     end
  end
```

Fig. 23. Procedure ArcCons for piecewise monotonic constraints.

```
procedure LocalArcCons(in i, j, in w, out Δ)
  begin
1   Δ := ∅;
2   kw := GroupOf(i, j, w);
3   if kw > last(j, i) then
4     begin
5       k := last(i, j);
6       while k > f(last(j, i)) do
7         begin
8           if not EmptyGroup(i, j, k) then
              Extend(i, j, k, Δ);
9           k := next(k)
10        end
11    end
  end
```

Fig. 24. Procedure LocalArcCons for piecewise monotonic constraints.

The definition of f_{ij} requires some sophistication to handle the case when S_k^{ij} (or S_k^{ji}) is unsupported. The above functions are assumed to take constant time to evaluate. As for monotonic constraints, subscripts are omitted in the algorithms presented in Figs. 23 and 24. Their correctness is an immediate consequence of the correctness of ArcCons and LocalArcCons for monotonic constraints. The complexity analysis is also similar to that for monotonic constraints. In all the executions of ArcCons and LocalArcCons for a given arc (i, j), a test in line 5 (ArcCons) or line 8 (LocalArcCons) is made at most once per group. Hence we have an optimal algorithm.

Theorem 29. *Algorithm AC-5 is* $O(ed)$ *for piecewise monotonic constraints.*

12. Application

We describe the application of AC-5 to constraint logic programming over finite domains. Constraint logic programming [9] is a class of languages whose main operation is constraint solving over a computation domain. A computation step amounts to checking the satisfiability of a conjunction of constraints.

Constraint logic programming over finite domains has been investigated in [23–25]. This is a computation domain in which constraints are equations, inequalities, and disequations over natural number terms or equations and disequations over constants. Natural number terms are constructed from natural numbers, variables ranging over a finite domain of natural numbers,

and the standard arithmetic operators ($+, \times, \ldots$). Some symbolic constraints are also provided to increase expressiveness and, in addition, users can define their own constraints. This computation domain is available in CHIP [5] and its constraint solver is based on consistency techniques, arithmetic reasoning, and branch and bound. It has been applied to numerous problems in combinatorial optimization such as graph coloring, warehouse location, scheduling and sequencing, cutting stock, assignment problems, and microcode labeling to name a few (see for instance [4, 24]).

Space does not allow us to present the operational semantics of the language. Let us just mention that the kernel of the constraint solver is an arc-consistency algorithm for a set of basic constraints. Other (non-basic) constraints are approximated in terms of the basic constraints and generate new basic constraints. The basic constraints are either *domain* constraints or *arithmetic* constraints, and are as follows (variables are represented by upper-case letters and constants by lower-case letters):

- domain constraint: $X \in \{a_1, \ldots, a_n\}$;
- arithmetic constraints: $aX \neq b$, $aX = bY + c$, $aX \leqslant bY + c$, $aX \geqslant bY + c$ with $a, a_i, b, c \geqslant 0$ and $a \neq 0$.

These constraints have been chosen carefully in order to avoid having to solve an NP-complete constraint satisfaction problem. For instance, allowing two variables in disequations or three variables in inequalities or equations leads to NP-complete problems.

We now show that AC-5 can be the basis of an efficient decision procedure for basic constraints.

Definition 30. A *system of constraints* S is a pair $\langle AC, DC \rangle$ where AC is a set of arithmetic constraints and DC is a set of domain constraints such that any variable occurring in an arithmetic constraint also occurs in some domain constraint of S.

Definition 31. Let $S = \langle AC, DC \rangle$ be a system of constraints. The set D_x is the *domain* of x in S (or in DC) iff the domain constraints of x in DC are $x \in D_1, \ldots, x \in D_k$ and D_x is the intersection of the D_i's.

Let us define a solved form for the constraints.

Definition 32. Let S be a system of constraints. S is in *solved form* iff any unary constraint $C(X)$ in S is node-consistent[8] with respect to the domain of X in S, and any binary constraint $C(X, Y)$ in S is arc-consistent with respect to the domains of X and Y in S.

[8] As usual, a unary constraint C is *node-consistent with respect to* D iff $\forall v \in D$: $C(v)$.

We now study a number of properties of systems of constraints in solved form.

Property 33. *Let* $C(X, Y)$ *be the binary constraint* $aX \leq bY + c$ *or* $aX \geq bY + c$, *arc-consistent with respect to* $D_X = \{v_1, \ldots, v_n\}$ *and* $D_Y = \{w_1, \ldots, w_m\}$. *Assume also that* $v_1 < \cdots < v_n$ *and* $w_1 < \cdots < w_m$. *Then we have that* C *is monotonic and* $C(v_1, w_1)$ *and* $C(v_n, w_m)$ *hold.*

Property 34. *Let* $C(X, Y)$ *be the binary constraint* $aX = bY + c$ *with* $a, b \neq 0$, *arc-consistent with respect to* $D_X = \{v_1, \ldots, v_n\}$ *and* $D_Y = \{w_1, \ldots, w_m\}$. *Assume also that* $v_1 < \cdots < v_n$ *and* $w_1 < \cdots < w_m$. *Then we have that* C *is functional,* $n = m$, *and* $C(v_i, w_i)$ *holds.*

The satisfiability of a system of constraints in solved form can be tested in a straightforward way.

Theorem 35. *Let* $S = \langle AC, DC \rangle$ *be a system of constraints in solved form.* S *is satisfiable if* $\langle \emptyset, DC \rangle$ *is satisfiable.*

Proof. It is clear that $\langle \emptyset, DC \rangle$ is not satisfiable iff the domain of some variable is empty in DC. If the domain of some variable is empty in DC, then S is not satisfiable. Otherwise, it is possible to construct a solution to S. By Properties 33 and 34, all binary constraints of S hold if we assign to each variable the smallest value in its domain. Moreover, because of node consistency, the unary constraints also hold for such an assignment. □

It remains to show how to transform a system of constraints into an equivalent one in solved form. This is precisely the purpose of the node- and arc-consistency algorithms.

Algorithm 36. To transform the system of constraints S into a system in solved form S':

(1) Apply a node-consistency algorithm to the unary constraints of $S = \langle AC, DC \rangle$ to obtain $\langle AC, DC' \rangle$.
(2) Apply an arc-consistency algorithm to the binary constraints of $\langle AC, DC' \rangle$ to obtain $S' = \langle AC, DC'' \rangle$.

Theorem 37. *Let* S *be a system of constraints. Algorithm 36 produces a system of constraints in solved form equivalent to* S.

We now give a complete constraint solver for the basic constraints. Given a system of constraints S, Algorithm 38 returns *true* if S is satisfiable and *false* otherwise.

Algorithm 38. To check the satisfiability of a system of constraints S:
(1) Apply Algorithm 36 to S to obtain $S' = \langle AC, DC \rangle$.
(2) If the domain of some variable is empty in DC, return *false*; otherwise return *true*.

In summary, we have shown that node- and arc-consistency algorithms provide us with a decision procedure for basic constraints. The complexity of the decision procedure is the complexity of the arc-consistency algorithm. Using the specialization of AC-5 for basic constraints, we obtain an $O(ed)$ decision procedure.

13. Discussion and related work

In this section, we discuss the practicability of our algorithms and their relationships with other work.

Our results indicate that many classes of constraints lead to an $O(ed)$ arc-consistency algorithm improving on the $O(ed^2)$ bound of [16]. Although a better asymptotic complexity does not guarantee a faster algorithm, empirical and theoretical results suggest the practicability of our results. On the theoretical side, it is easy to see that the constant factors are in fact small in our algorithms (in general 1 or 2). On the empirical side, most of these classes have been integrated in the `cc(FD)` programming language [26] improving the computational results of many algorithms compared to the previous versions based on AC-3 and AC-4. This will be discussed in a forthcoming paper. It is however important to note that AC-4 and some classes studied here increase the memory requirement. Hence, for memory management reasons, AC-3 may sometimes be preferable.

As far as related work is concerned, three closely related papers deserve to be mentioned. Mohr and Masini [17] also discovered independently the subset of arithmetic constraints that can be solved in $O(ed)$. The constraints considered were binary equations, inequalities, and disequations, which are respectively subcases of functional, monotonic, and anti-functional constraints. They indicate informally how to modify AC-4 to include these constraints, but do not present a uniform and generic algorithm like AC-5.

Perlin's algorithm [21] is an arc-consistency algorithm working on a graph representation of the CSP where the values (not the variables) are nodes and the constraints are represented by links between nodes. The algorithm is then bounded by the size of the graph. Perlin investigates the idea of factoring constraints in this graph representation. More precisely, he studies the idea of splitting a constraint $C(x, y)$ into a conjunction of three constraints $C_1(x, t_1)$ & $C_2(t_1, t_2)$ & $C_3(t_2, y)$ (with t_1 and t_2 being two new variables) such that

(1) arc consistency produces the same pruning on the problem variables;
(2) the graph associated to the new problem is smaller than the initial graph.

It turns out that arc consistency runs in $O(ed)$ when the constraints all express equalities between some of the constraint variables. Note that, in this case, $C_2(t_1, t_2)$ reduces to an equation (a subcase of functional constraints). The contributions of Perlin can thus be summarized as

(1) the identification of a general preprocessing technique, factorization, to reduce the size of the graph; and
(2) the identification of a special kind of functional constraints.

It should be easy to generalize those results to the case of functional constraints between some of the constraint variables. Similarly, we believe (but have not yet proven) that the bound for piecewise monotonic constraints can be obtained from factorization, piecewise functional constraints, and monotonic constraints. Note however, that an inconvenience of the graph representation is its memory requirement: a functional constraint requires $O(d)$ space with the graph representation and requires constant space in AC-5.

Arc consistency of functional constraints can be solved through a reduction to 2-sat [10], keeping the $O(ed)$ result. However, this algorithm also uses $O(d)$ space per constraint.

Finally, it is also interesting to study the evolution of arc-consistency algorithms. The main contribution of AC-4 was the idea of working with domain values instead of domain variables. This idea is systematically exploited by Perlin to obtain a better bound for some classes of constraints through factorization. Exploiting the structure of the domains is the new idea behind Mohr and Masini's work and the monotonic constraints of this paper. Finally, exploiting the structure of the constraints is the key idea behind the piecewise constraints of this paper. AC-5 accommodates these results in a unified and generic algorithm.

14. Conclusion

A new generic arc-consistency algorithm AC-5 is presented whose specializations include, not only AC-3 and AC-4, but also an $O(ed)$ algorithm for important subclasses of constraints including functional, monotonic, and anti-functional constraints, as well as their piecewise counterparts. An application of AC-5 to constraint logic programming over finite domains is described. Together with node consistency, it provides the main algorithms for an $O(ed)$ decision procedure for basic constraints. From a software engineering perspective, AC-5 has the advantage of uniformity. Each constraint may have a particular implementation, based on AC-3, AC-4, or some specific techniques,

without influencing the main algorithm. As a consequence, many different implementation techniques can be interleaved together in a natural setting.

Current research is devoted to applying these ideas to path-consistency and non-binary constraints. It turns out that similar improvements can be obtained for path-consistency algorithms although the algorithms are somewhat more complicated. Non-binary constraints are also being investigated to obtain the equivalent of GAC-4 [15] for AC-5. Preliminary results indicate that the results carry over for some classes of constraints, although once again the algorithms are more involved.

Acknowledgement

We thank an anonymous IJCAI reviewer for mentioning the reduction to 2-sat, Eugene Freuder for pointing out the work of Perlin, and the anonymous *AI Journal* reviewers for their careful comments and suggestions. The help of Trina Avery for correcting our English is also appreciated. This research was supported in part by the National Science Foundation under grant number CCR-9108032 and by the Office of Naval Research under grant N00014-91-J-4052, ARPA order 8225.

References

[1] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990).

[2] R. Dechter and J. Pearl, Network-based heuristics for constraint satisfaction problems, *Artif. Intell.* **34** (1988) 1–38.

[3] Y. Deville and P. Van Hentenryck, An efficient arc consistency algorithm for a class of CSP problems, in: *Proceedings IJCAI-91*, Sydney, Australia (1991).

[4] M. Dincbas, H. Simonis and P. Van Hentenryck, Solving large combinatiorial problems in logic programming, *J. Logic Programming* **8** (1–2) (1990) 75–93.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The constraint logic programming language CHIP, in: *Proceedings International Conference on Fifth Generation Computer Systems*, Tokyo, Japan (1988).

[6] E.C. Freuder, Synthesizing constraint expressions, *Commun. ACM* **21** (1978) 958–966.

[7] J. Gaschnig, A constraint satisfaction method for inference making, in: *Proceedings 12th Annual Allerton Conference on Circuit System Theory*, Urbana-Champaign, IL (1974) 866–874.

[8] R.M. Haralick and G.L. Elliot, Increasing tree search efficiency for constraint satisfaction problems, *Artif. Intell.* **14** (1980) 263–313.

[9] J. Jaffar and S. Michaylov, Methodology and implementation of a CLP system, in: *Proceedings Fourth International Conference on Logic Programming*, Melbourne, Australia (1987).

[10] S. Kasif, On the parallel complexity of discrete relaxation in constraint satisfaction networks, *Artif. Intell.* **45** (1990) 275–286.

[11] J.-L. Lauriere, A language and a program for stating and solving combinatorial problems, *Artif. Intell.* **10** (1) (1978) 29–127.

[12] A.K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1) (1977) 99–118.

[13] A.K. Mackworth, *Constraint Satisfaction* (Wiley, New York, 1987).
[14] A.K. Mackworth and E.C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artif. Intell.* **25** (1985) 65–74.
[15] R. Mohr, Good old discrete relaxation, in: *Proceedings ECAI-88*, Munich, Germany (1988).
[16] R. Mohr and T.C. Henderson, Arc and path consistency revisited, *Artif. Intell.* **28** (1986) 225–233.
[17] R. Mohr and G. Masini, *Running Efficiently Arc Consistency* (Springer, Berlin, 1988) 217–231.
[18] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Inf. Sci.* **7** (2) (1974) 95–132.
[19] U. Montanari and F. Rossi, An efficient algorithm for the solution of hierarchical networks of constraints, in: *Workshop on Graph Grammars and Their Applications in Computer Science*, Warrenton (1986).
[20] B. Nadel, Constraint satisfaction algorithms, *Comput. Intell.* **5** (4) (1989) 188–224.
[21] M. Perlin, Arc consistency for factorable relations, in: *Proceedings, Third International Conference on Tools for Artificial Intelligence*, San Jose, CA (1991) 340–345.
[22] R.E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Discrete Methods* **6** (1985) 306–318.
[23] P. Van Hentenryck, A framework for consistency techniques in logic programming, in: *Proceedings IJCAI-87*, Milan, Italy (1987).
[24] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming*, Logic Programming Series (MIT Press, Cambridge, MA, 1989).
[25] P. Van Hentenryck and M. Dincbas, Domains in logic programming, in: *Proceedings AAAI-86*, Philadelphia, PA (1986).
[26] P. Van Hentenryck, V. Saraswat and Y. Deville, Constraint processing in `cc(FD)`, Tech. Rept. Brown University, Providence, RI (1992).
[27] D. Waltz, Generating semantic descriptions from drawings of scenes with shadows, Tech. Rept. AI271, MIT, Cambridge, MA (1972).