A Generic Arc Consistency Algorithm and its Specializations

Pascal Van Hentenryck[1]
Yves Deville[2]
Choh-Man Teng[3]

Technical Report No. CS-91-65
December, 1991

[1]Brown University, Computer Science Department, Box 1910, Providence, RI 02912 USA
[2]Université Catholique de Louvain, Pl. Ste Barbe 2, B-1348 Louvain-La-Neuve, Belgium
[3]Brown University, Computer Science Department, Box 1910, Providence, RI 02912

A Generic Arc Consistency Algorithm and its Specializations[1]

Pascal Van Hentenryck[2], Yves Deville[3] and Choh-Man Teng[2]

Abstract

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 [25] and culminating in the optimal algorithm AC-4 of Mohr and Henderson [15]. 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. The 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 [23]. The kernel of the constraint-solver for such a programming 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 $\mathcal{NP}$-complete which mean that backtracking search is an important technique for solving them.

Many search algorithms (e.g. [2, 6, 7, 8, 11, 18]), preprocessing techniques and constraint algorithms (e.g. [25, 17, 12, 14, 15]) have been designed and analyzed for this class of problems. See the reviews [13, 19] for a comprehensive overview of this area. In this paper,

[1] This paper is an extended version of [3].

[2] Brown University, Box 1910, Providence, RI 02912, USA

[3] Université Catholique de Louvain, Pl. Ste Barbe 2, B-1348 Louvain-La-Neuve, Belgium

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 sure that the constraints are individually consistent. Arc consistency algorithms have a long story on their own. They originates from Waltz filtering algorithm [25] and were refined several times [12] to culminate in the optimal algorithm AC-4 of Mohr and Henderson [15]. 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[5] been applied in the design of Constraint Logic Programming (CLP) languages, more precisely in the design and implementation of CHIP [23, 5]. CHIP allows for 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 preserving the efficiency of imperative languages, yet shortening the development time significantly. Examples of applications include graph-coloring, warehouse locations, car-sequencing and cutting stock (see for instance [4, 23]). 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.

This research originated as an attempt to improve further the efficiency of the kernel algorithm. This paper contains two contributions.

First 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 $O(ed)$.

The rest of this paper is organized in the following way. Section 2 fixes 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, 6, and 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, 10 and 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 contains the conclusion of this research.

[5] Although Mackworth already mentioned as early as 1977 [12] the potential value of consistency techniques for programming languages.

2 Preliminaries

To describe the CSP, 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 $(i < j)$, there is at most one constraint relating them. This constraint is 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 D the union of all domains and 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} while 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 wrt* D_i *and* D_j iff $\forall v \in D_i$, $\exists w \in D_j : C_{ij}(v, w)$.

Definition 2 Let $\mathcal{P}$ be $D_1 \times \ldots \times D_n$. A graph G is *arc consistent wrt* $\mathcal{P}$ iff $\forall\, (i, j) \in arc(G)$: (i, j) is arc consistent wrt D_i and D_j.

The next definition is useful to specify the outcome of an arc consistent algorithm.

Definition 3 Let $\mathcal{P}$ be $D_1 \times \ldots \times D_n$. Let $\mathcal{P}' \subseteq \mathcal{P}$. G is a *maximally arc consistent wrt* P' *in* $\mathcal{P}$ iff G is *arc consistent wrt* $\mathcal{P}'$ and there is no other $\mathcal{P}''$ with $\mathcal{P}' \subset \mathcal{P}'' \subseteq \mathcal{P}$ such that G is arc consistent wrt $\mathcal{P}''$.

The purpose of an arc consistency algorithm is to compute, given a graph G and a set P, a set P' such that G is maximally arc consistent wrt P' in 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 in AC-5 is to have a queue containing elements $< (i, j), w >$ where (i, j) is an arc and w is a value which has been removed from D_j and justifies the need to reconsider arc (i, j). As a consequence, AC-5 can be specialized to obtain either AC-3 or AC-4 by giving a particular implementation of Procedures ArcCons and LocalArcCons. Moreover, for certain class of constraints, AC-5 can be specialized to give an $O(ed)$ algorithm.

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.

procedure INITQUEUE(**out** Q)
Post: $Q = \{\}$.
function EMPTYQUEUE(**in** Q): Boolean
Post: EMPTYQUEUE $\Leftrightarrow$ ($Q = \{\}$).
procedure DEQUEUE(**inout** Q, **out** i, j, w)
Post: $< (i,j), w > \in Q_0$ and $Q = Q_0 \setminus < (i,j), w >$.
procedure ENQUEUE(**in** i, Δ, **inout** Q)
Pre: $\Delta \subseteq D_i$ and $i \in node(G)$.
Post: $Q = Q_0 \cup \{< (k,i), v > | \, (k,i) \in arc(G)$ and $v \in \Delta\}$.

Figure 1: The QUEUE Module

3.1 Operations on Queues

The operations we need are described in Figure 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 when the set of values Δ has been removed from D_i. It introduces elements of the form $< (k,i), v >$ 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 but 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) and a lazy distribution of v on the arcs. To achieve this result, the queue could be organized to contain elements of the form $< \{A_1, \ldots, A_m\}, v >$ where A_k is an arc and v is a value. Procedure ENQUEUE(i, Δ, Q) adds an element $< \{A_1, \ldots, A_m\}, v >$ 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 $< \{A_1, \ldots, A_m\}, w >$ with $m > 0$, remove an $A_k = (i,j)$ from the set, and returns i, j, and w.

3.2 Specification of the Parametric Procedures

Figure 2 gives the specification of the two subproblems. Their implementations for various kinds of constraints will be 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 for the result Δ to be returned. It is sufficient to compute Δ_1 to guarantee the correctness of AC-5. However the procedure gives the opportunity to achieve more pruning (up to Δ_2) while still preserving

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 $\forall w' \in D_j : \neg C_{ij}(v,w')\}$,
$\Delta_2 = \{v \in D_i \mid \forall w' \in D_j : \neg C_{ij}(v,w')\}$.

Figure 2: Specification of the Procedures

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.

3.3 Algorithm AC-5

We are now in position to present Algorithm AC-5. The algorithm is depicted in Figure 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. For ease of presentation, we assume that AC-5 stops (with the answer $\mathcal{P}$ empty) as soon as a domain becomes empty. The second step applies LOCALARCCONS on each of the element of the queue possibly generating new elements in the queue. The correctness of the algorithm is an immediate consequence of the correctness of Algorithm AC-3 [12] that it generalizes. AC-3 is a particular case of AC-5 where the value w is never used in the implementation of Procedure LOCALARCCONS. AC-4 is a particular case of AC-5 where the implementation of Procedure LOCALARCCONS does not use node i. Of course, the data-structures used in AC-4 are more sophisticated than those of AC-3.

In order to prove various results on AC-5, 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 Figure 4. Note that the actual implementation does not need to perform these operations. They are just presented here to ease the presentation and simplify the theorem.

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

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

We are now in position to prove the following theorem.

Theorem 4 Algorithm AC-5 has the following three 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) If $s_1, \ldots, s_p$ are the number of new elements in the queue on each iteration at line 12, then $s_1 + \ldots + s_p \leq O(ed)$.

```
Algorithm AC-5
Post: let P_0 = D_{1_0} × ... × D_{n_0},
          P = D_1 × ... × D_n
      G is maximally arc consistent wrt P in P_0.
   begin AC-5
1     InitQueue(Q)
2     for each (i,j) ∈ 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
```

Figure 3: The Arc Consistency Algorithm AC-5

```
procedure InitQueue(out Q)
Post: ∀ (k,i) ∈ arc(G) : Status[(k,i),v] = present if v ∈ D_i
                                          = rejected if v ∉ D_i
function EmptyQueue(in Q)
Post: ∀ (k,i) ∈ arc(G) ∀v : Status[(k,i),v] ≠ suspended.

procedure Dequeue(inout Q, out i, j, w)
Post: Status[(i,j),w] = rejected.

procedure Enqueue(in i, Δ, inout Q)
Pre: ∀ (k,i) ∈ arc(G) ∀v ∈ Δ : Status[(k,i),v] = present.
Post: ∀ (k,i) ∈ arc(G) ∀v ∈ Δ : Status[(k,i),v] = suspended.
```

Figure 4: The QUEUE Module on Structure *Status*

Proof

Property 1 holds initially. Assuming that it holds in line 2, it also holds after an iteration of lines 4 to 6. Line 5 makes sure that $< (j,i), v >$ is suspended for all $v \in \Delta$ and put 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 to 13 preserves the invariant. Lines 10 and 11 maintain it on their own. Lines 12 and 13 respectively make sure that $< (j,i), v >$ is rejected for all $v \in \Delta$ and remove Δ from D_i.

Property 2 holds because each element of *Status* is only allowed to make 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.

Property 3 is a direct consequence 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 of the size of the domains. The space complexity of AC-5 is $O(ed + nd)$. The above theorem can be used to deduce the overall complexity of AC-5 from the complexity of Procedures ARCCONS and LOCALARCCONS.

Theorem 5 (1) If the time complexity of ARCCONS is $O(d^2)$, then, the time complexity of AC-5 is at least $O(ed^2)$. (2) If the time complexity of ARCCONS is $O(d)$ and the time complexity of LOCALARCCONS(i,j,w,Δ) is $O(\Delta)$[6], then the time complexity of AC-5 is $O(ed)$.

In particular, in AC-3 and AC-4, Procedure ARCCONS is necessarily $O(d^2)$, hence an overall complexity of at least $O(ed^2)$. There is no other possibility to reduce the complexity than considering particular classes of constraints, allowing to implement, in particular, Procedure ARCCONS in $O(d)$. Note also that an AC algorithm in $O(ed)$ will be optimal for a subclass of constraints since it is reasonable to assume that we need to check at least once each value in each domain. 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 $O(ed)$.

4 Representation of Domains

Particular implementations of ARCCONS and LOCALARCCONS will perform operation on domains that are depicted in Figure 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 ordering on the domain D for reasons that

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

function SIZE(**in** D): Integer
Post: SIZE $=\mid D \mid$.
procedure REMOVEELEM(**in** v, **inout** D)
Post: $D = D_0 \setminus \{v\}$.
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

Figure 5: The DOMAIN module

will become clear later.[7] 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 Figure 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 take care updating all fields to preserve the semantics. This can be done in constant time.

When the domain is sparse, the representation depicted in Figure 7 can be used. It reasons about indices instead of values and use an hash-table to test membership to the domain. Although the time complexity of membership is theoretically not $O(1)$, under reasonable assumption, the expected time to search for an element is $O(1)$ [1].

5 Functional Constraints

Definition 6 A constraint C is *functional* wrt a domain D iff for all v (resp. w) $\in D$ there exists at most one w (resp. 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 functional constraint is $x = y + 5$.

[7]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 $S = \{b, \ldots, B\}$
$D_i = \{v_1, \ldots, v_m\} \subseteq S$ 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 $v \in D_i$
$D_i.succ[v_k] = v_{k+1}$ $(1 \leq k < m)$
$D_i.succ[v_m] = +\infty$
$D_i.pred[v_{k+1}] = v_k$ $(1 \leq k < m)$
$D_i.pred[v_1] = -\infty$

Figure 6: DOMAIN Data Structure: Consecutive Values.

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 $\in \{1, \ldots, a\}$
$D_i.max$: integer $\in \{1, \ldots, a\}$
$D_i.element$: 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
$D_i.value[v] = e_v$
$D_i.succ[v_k] = v_{k+1}$ $(1 \leq k < m)$
$D_i.succ[v_m] = +\infty$
$D_i.pred[v_{k+1}] = v_k$ $(1 \leq k < m)$
$D_i.pred[v_1] = -\infty$

Figure 7: DOMAIN Data Structure: Sparse Values.

```
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
```

Figure 8: ArcCons for Functional Constraints

```
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
```

Figure 9: LocalArcCons for Functional Constraints

Convention 7 If C_{ij} is a functional constraint, we denote by $f_{ij}(v)$ (resp. $f_{ji}(w)$) the value w (resp. v) such that $C_{ij}(v, w)$. If such a value does not exist, the function will denote a value outside the domain for which the constraint holds.

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 are now in position to present Procedures ArcCons and LocalArcCons for functional constraints. They are depicted in Figures 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 8 Algorithm AC-5 is $O(ed)$ for functional constraints wrt D.

Note that functional constraints do not add any requirement for the basic operations on the domains compared to traditional algorithms.

```
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
```

Figure 10: Procedure ArcCons for Anti-Functional Constraints

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

Figure 11: Procedure LocalArcCons for Anti-Functional Constraints

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.

Definition 9 A constraint C_{ij} is *anti-functional* wrt a domain D iff $\neg C_{ij}$ is functional wrt 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. The procedures ArcCons and LocalArcCons are shown in Figures 10 and 11. We use the same convention as for functional constraints.

Instead of considering each element of D_i, hence with a complexity of $O(d)$, 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 this 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.[8]

Theorem 10 Algorithm AC-5 is $O(ed)$ for anti-functional constraints wrt D.

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

```
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        Δ := Δ ∪ {v};
6        v := next(v, D_i)
7     end
   end
```

Figure 12: ArcCons for Monotonic Constraints

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

Figure 13: LocalArcCons for Monotonic Constraints

7 Monotonic Constraints

We now consider another class of constraints: monotonic constraint. An example of monotonic constraint is $x \leq y - 3$. 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 exists elements w_1, 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 11 A constraint C is *monotonic wrt* D iff there exists a total ordering on D such that, for any value v, w in D, $C(v, w)$ holds implies $C(v', w')$ holds for all values v', w' in D such that $v' \leq v$ and $w' \geq w$.

Convention 12 Since AC-5 is working with arcs, we associate to each arc (i, j) three functions f_{ij}, $last_{ij}$, and $next_{ij}$ and a relation $\succ_{ij}$. Given a monotonic constraint C_{ij}, the functions and relation for arc (i, j) are as follows $f_{ij}(w) = max\{v \mid C_{ij}(v, w)\}$, $last_{ij} =$ Max, $next_{ij} =$ Pred, $\succ_{ij} = >$ while those for arc (j, i) are $f_{ji}(v) = min\{w \mid C_{ij}(v, w)\}$, $last_{ji} =$ Min, $next_{ji} =$ Succ, $\succ_{ji} = <$.
Moreover, since Procedures ArcCons and LocalArcCons only use f_{ij}, $last_{ij}$, $next_{ij}$, and $\succ_{ij}$ 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 position to describe the implementation of Procedures ArcCons and LocalArcCons for monotonic constraints. They are depicted in Figures 12 and 13.

```
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
```

Figure 14: Revised Procedure LocalArcCons for Monotonic Constraints

Lemma 1 Procedures ArcCons and LocalArcCons fulfill their specifications.

Proof Procedure ArcCons and LocalArcCons compute the set $\Delta = \{v \in D_i \mid v \succ f(last(D_j))\}$. By monotonicity of the constraint, $\Delta \subseteq \Delta_2$ with $\Delta_2 = \{v \in D_i \mid \forall w' \in D_j : \neg C_{ij}(v, w')\}$, and $\Delta_2 \cap \{v \in D_i \mid v \preceq f(last(D_j))\} = \emptyset$. Hence $\Delta = \Delta_2$ and both postconditions are satisfied. □

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

Theorem 13 Procedure AC-5 is $O(ed)$ for monotonic constraints wrt D.

It is also clear that AC-5 can be applied at the same time to (anti-)functional and monotonic constraints keeping 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}), hence eliminating the need to update part of the data structure. The revised procedure ArcCons is depicted in Figure 14. The only difference lies in lines 5 and 6. This has obviously no influence on the correctness of ArcCons.

Procedure LocalArcCons could use ArcCons, but a revised version is presented in Figure 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 $w \preceq last(D_j)$, then Δ_1 is empty by the monotonicity of C_{ij}. It is possible to compute Δ_1[9], but this would prevent the reduction of domains as soon as possible.

Theorem 14 With the revised implementation depicted in Figures 14 and 15, Procedure AC-5 is $O(ed)$ for monotonic constraints wrt D.

[9]Replace $f(last(D_j))$ by $f(w)$ in line 4 in Figure 15.

```
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
```

Figure 15: Revised Procedure LocalArcCons for Monotonic Constraints

Proof This proof requires the use of amortized complexity [21] 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 which 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 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 $v' \succ last(D_i)$ and this value will thus never be considered anymore. This hold because 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 thus still have an optimal AC-5 algorithm. □

8 Piecewise Constraints

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

Convention 15 Let S, P be sets, and C be a constraint. $C(S,P)$ denotes $\forall v \in S, \forall w \in P : C(v,w)$. $\neg C(S,P)$ denotes $\forall v \in S, \forall w \in P : \neg C(v,w)$. We also use $C(S,w)$ for $C(S,\{w\})$.

Definition 16 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 wrt C iff for all $S_k \in \mathcal{S}, P_{k'} \in \mathcal{P} : C(S_k,P_{k'})$ or $\neg C(S_k,P_{k'})$ holds.

function NBGROUP(**in** i, j): Integer
Post: NBGROUP $=\mid \mathcal{S}_{ij} \mid -1$.
function SIZEOFGROUP(**in** i, j, k): Integer
Pre: $0 \leq k \leq$ NBGROUP(i,j)
Post: SIZEOFGROUP $= \mid S_k^{ij} \cap D_i \mid$
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: $\Delta = \Delta_0 \cup (S_k^{ij} \cap D_i)$
$\quad$ *Status-pd*$[(i,j),k]$ = **true**
function GROUPOF(**in** i, j, v): Integer
Pre: $v \in D_i^{init}$
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 $= \mid \{k \mid S_k^{ij} \cap D_i \neq \emptyset\} \mid$

Figure 16: The PIECEWISE DECOMPOSITION module.

Representation of Piecewise Constraints

Before presenting implementation of ARCCONS and LOCALARCCONS for constraints having some particular piecewise decomposition, operations on piecewise decompositions are depicted in Figure 16. For ease of implementation, we assume that elements in groups of a piecewise decomposition are never removed during the execution. The piecewise decomposition of D_i and D_j wrt 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:

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

Thus, *Status-pd* has to be **false** when the corresponding group is not empty. The primitive operations on a piecewise decomposition are assumed to take constant time, except for EXTEND which complexity is assumed to be $O(s)$, where s is the size of S_k^{ij}.

A simple data structure that enables to achieve these results is given if Figure 17. Its space complexity is $O(d)$ per piecewise decomposition. This data structure cannot be updated by the REMOVEELEM primitive in a constant time since an element in a domain can belong to different groups in different piecewise decompositions. The update can however easily be performed by the ENQUEUE primitive without affecting its complexity.

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

Syntax

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

Semantics

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

Figure 17: PIECEWISE DECOMPOSITION Data Structure.

It is not difficult to initialize the data structure in $O(d)$ given the realistic assumption that it takes $O(s)$ to find the s elements in D_j (resp. D_i) supporting a value v (resp. w) in D_i (resp. 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 which domain can be decomposed into groups such that each group of D_i (resp. D_j) is supported by at most one group of D_j (resp. D_i).

Definition 17 A constraint C_{ij} is *piecewise functional* wrt D_i, 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 wrt C_{ij} such that for all $S_k \in \mathcal{S}$ (resp. $P_{k'} \in \mathcal{P}$), there exists at most one $P_{k'} \in \mathcal{P}$ (resp. $S_k \in \mathcal{S}$), such that $C_{ij}(S_k, P_{k'})$.

Examples of functional piecewise constraints are the modulo ($x = y \bmod z$) and integer division ($x = y \text{ div } z$) constraints. The `element` constraint of the CHIP programming language [23] is a piecewise constraint as well. Finally note that functional constraints are a subclass of piecewise constraints, where 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 (resp. D_j) are in the same group (e.g. S_0 and P_0), then the piecewise decomposition

$\mathcal{S} = \{S_0, \ldots, S_n\}$ and $\mathcal{P} = \{P_0, \ldots, P_n\}$ have the same number of groups, and groups can be renumbered such that the following holds:

(PF1) $\neg C_{ij}(S_0, D_j)$ and $\neg C_{ij}(D_i, P_0)$

(PF2) $C_{ij}(S_k, P_k) \quad (1 \leq k \leq n)$

(PF3) $\neg C_{ij}(S_k, P_{k'}) \quad (1 \leq k, k' \leq n \text{ and } k \neq k')$

The implementation of ARCCONS and LOCALARCCONS for piecewise functional constraints will assume a piecewise decomposition respecting PF1–3. The following property states necessary and sufficient conditions to have a piecewise functional constraint.

Property 18 A constraint C_{ij} is piecewise functional wrt 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 $w \in D_j$ and $0 \leq k, k' \leq n$ and $k \neq k'$)

Proof The "only if" part is straightforward. For the "if" part, let us assume the existence of some unsupported element in D_i and in D_j, and that all the unsupported element 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.

$$\begin{aligned} P_k &= \{w \in D_j | \exists v \in S_k \;\; C_{ij}(v, w)\} \quad (1 \leq k \leq n) \\ P_0 &= D_j \setminus \bigcup_{1 \leq l \leq n} P_l \end{aligned}$$

It is sufficient to prove that $\mathcal{P}$ is a partition and that $\mathcal{S}$ and $\mathcal{P}$ satisfy PF1–3.

($\mathcal{P}$ is a partition). (A) $P_k \cap P_{k'} = \emptyset$ $(k \neq k')$. This hold 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 $P_k \neq \emptyset$.

(PF1). Hold 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 thus 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 constraint are given in Figures 19 and 20. Line 2 handles the group S_0^{ij} containing all the unsupported elements of the initial domain D_i. They use the boolean function UNSUPPORTED specified in Figure 18. The correctness of these procedures is a 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: $0 \leq k \leq \text{NBGROUP}(i,j)$
Post: UNSUPPORTED $\Leftrightarrow$ EMPTYGROUP$(j,i,k) \wedge \neg$ *Status-pd*$[(i,j),k]$

Figure 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
```

Figure 19: ARCCONS 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
```

Figure 20: LOCALARCCONS for Piecewise 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
```

Figure 21: Procedure ArcCons for Piecewise Anti-Functional Constraints

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

Figure 22: Procedure LocalArcCons for Piecewise Anti-Functional Constraints

The time complexity is analyzed globally within AC-5. If the complexity of all the execution 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 depend 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, hence a complexity bounded by $O(d)$. Using amortized complexity as in the case of monotonic constraints, it follows that we have an optimal algorithm.

Theorem 19 Procedure AC-5 is $O(ed)$ for piecewise functional constraints wrt D.

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 which domain D_i and D_j can be decomposed in groups such that each group of D_i (resp. D_j) is not supported by at most one group of D_j (resp. D_i).

Definition 20 A constraint C_{ij} is *anti-functional* wrt D_i. D_j iff $\neg C_{ij}$ is piecewise functional wrt D_i, D_j.

Using the same notations as in the preceding section, procedures ArcCons and LocalArcCons for anti-functional constraint can easily be extended in the piecewise framework (see Figures 21 and 22). Note the test for $k \neq 0$ since group 0 supports all groups. By

```
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
```

Figure 23: Procedure LocalArcCons for Piecewise Monotonic Constraints

a complexity analysis similar to 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 21 Algorithm AC-5 is $O(ed)$ for piecewise anti-functional constraints.

11 Piecewise Monotonic Constraints

Monotonic constraint are finally generalized to piecewise monotonic constraints. An example of piecewise monotonic constraint is $x \leq y$ div 5.

Definition 22 A constraint C_{ij} is *piecewise monotonic* wrt D_i, 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 wrt 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 23 As for monotonic constraint, we associate to each arc (i, j) three functions f_{ij}, $last_{ij}$, and $next_{ij}$ and a relation $\succ_{ij}$. Given a piecewise monotonic constraint C_{ij}, the functions and relation for arc (i, j) are as follows: $f_{ij}(k) = max\{\{-1\} \cup \{k' \mid C_{ij}(S_k^{ij}, S_{k'}^{ji})\}\}$, $last_{ij}(a, b) =$ LastGroup(a, b), $next_{ij}(k) = k - 1$, $\succ_{ij} = >$ while those for arc (j, i) are $f_{ji}(k) = min\{\{$ NbGroup$(j, i)+1\} \cup \{k' \mid C_{ij}(S_{k'}^{ij}, S_k^{ji})\}\}$, $last_{ji}(a, b) =$ FirstGroup(a, b), $next_{ji}(k) = k + 1$, $\succ_{ji} = <$.

The definition of f_{ij} requires some sophistication to capture 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 will be omitted in the algorithms presented in Figures 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 the one made for monotonic constraints. In all the execution 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 24 Algorithm AC-5 is $O(ed)$ 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            v := next(k)
10        end
11     end
   end
```

Figure 24: Procedure LocalArcCons 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 step of computation amounts to check the satisfiability of a conjunction of constraints.

Constraint Logic Programming over finite domains has been investigated in [24, 22, 23]. It is a computation domain where 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 ($+$, $\times$, ...). Also some symbolic constraints are provided to increase the expressiveness and the user has the ability to define its own constraints. This computation domain is available in CHIP [5] and its constraint-solver is based on consistency techniques, arithmetic reasoning, and branch & bound. It has been applied to numerous applications 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, 23].

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 *arithmetics* 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: $aX \neq b$, $aX = bY + c$, $aX \leq bY + c$, $aX \geq bY + c$ with

$a, a_i, b, c \geq 0$ and $a \neq o$.

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

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

Definition 25 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 26 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 27 Let S be a system of constraints. S is in *solved form* iff any unary constraint $C(X)$ in S is node consistent[10] wrt the domain of X in S, and any binary constraint $C(X, Y)$ in S is arc consistent wrt the domains of X, Y in S.

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

Property 28 Let $C(X, Y)$ be the binary constraint $aX \leq bY + c$ or $aX \geq bY + c$, arc consistent wrt $D_X = \{v_1, \ldots, v_n\}, D_Y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have C is monotonic and $C(v_1, w_1)$ and $C(v_n, w_m)$ hold.

Property 29 Let $C(X, Y)$ be the binary constraint $aX = bY + c$ with $a, b \neq 0$, arc consistent wrt $D_X = \{v_1, \ldots, v_n\}, D_Y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have 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 30 Let $S = \langle AC, DC \rangle$ be a system of constraints in solved form. S is satisfiable iff $\langle \emptyset, DC \rangle$ 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 28 and 29, 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.

[10] As usual, a unary constraint C is *node consistent wrt* D iff $\forall v \in D : C(v)$.

Algorithm 31 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 32 Let S be a system of constraints. Algorithm 31 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 33 returns *true* if S is satisfiable and $false$ otherwise.

Algorithm 33 To check the satisfiability of a system of constraints S: (1) apply Algorithm 31 to S to obtain $S' = \langle AC, DC \rangle$ and (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 Related Work

Arc-consistency of functional constraints can be solved through a reduction to 2-sat [10], keeping the O(ed) result. However, this algorithm uses space linear in the size of the domains for each constraint while our algorithm uses constant space per constraint. Perlin [20] discusses independently a particular case of piecewise constraints. Mohr and Mansini [16] have also discovered independently that the simple arithmetic constraints of CHIP can be made arc-consistent in time $O(ed)$.

14 Conclusion

A new generic arc consistency algorithm AC-5 has been presented whose specializations include, not only AC-3 and AC-4, but also an $O(ed)$ algorithms for an important subclass of networks containing functional and monotonic constraints. An application of AC-5 to Constraint Logic Programming over finite domains has been 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.

Future research on this topic includes the search for other subclasses whose properties allow for an $O(ed)$ algorithm. Path consistency has not been considered seriously in CLP languages and generalizations of the above ideas to path consistency and support for path consistency in CLP languages deserve future attention.

Acknowledgments

Thanks to an anonymous IJCAI reviewer for mentioning the reduction to 2-sat and to Eugene Freuder for pointing out the work of Perlin. 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.