An Efficient Arc Consistency Algorithm for a Class of CSP Problems

Yves Deville*
University of Namur, 21 rue Grandgagnage
B-5000 Namur (Belgium)
Email: yde@infoiundp.ac.be

Pascal Van Hentenryck
Brown University, Box 1910,
Providence, RI 02912
Email: pvh@cs.brown.edu

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 [20] and culminating in the optimal algorithm AC-4 of Mohr and Henderson [13]. AC-4 runs in $0(ed^2)$ in the worst case where e is the number of arcs (or constraints) and *d* is the site 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 parametrised 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 *0(ed)* algorithm for two important classes of constraints: functional and monotonic constraints.

We also show that AC-5 has an important application in Constraint Logic Programming over Finite Domains [18]. 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 *0(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 mean that backtracking search is an important technique for solving them.

Many search algorithms (e.g. [I, 4, 5, 6, 9, 15]), preprocessing techniques and constraint algorithms (e.g. [20, 14, 10, 12, 13]) have been designed and analysed for this class of problems. See the reviews 111, 16] 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 sure that the constraints are individually consistent. Arc consistency algorithms have a long story on their own. They originates from Waltz filtering algorithm [20] and were refined several times [10] to culminate in the optimal algorithm AC-4 of Mohr and Henderson [13]. AC-4 runs in $0(ed^2)$ 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 [18, 3]. 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 [2, 18]). The kernel of CHIP for finite domainB 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 parametrised 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 specialised to produce an *0(ed)* arc consistency algorithm for two important classes of constraints; functional and monotonic constraints.

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

* Supported by the Belgian National Fund for Scientific Research as a Research Associate.

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

decision procedure for the basic constraints running in time *0(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. Sections 4 and 5 show how an *0(ed)* algorithm can be achieved for functional and monotonic constraints by giving particular implementations of the two procedures. Section 6 presents various representations for the domains while Section 7 shows that AC-5, in conjunction with node consistency, provides an *0(ed)* decision procedure for the basic constraints of CLP over finite domains. Section 8 contains the conclusion of this research.

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 $\mathcal{P}' \subseteq \mathcal{P}$. $\mathcal{G}$ 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 ACM 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), ID > where (i,*j)* is an arc and *w* is a value which has been removed from *Dj* 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 *0(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 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** $v \in \Delta\}$.
procedure DEQUEUE(**inout** Q, **out** i, j, w)
Post: $< (i, j), w > \in Q_0$ **and** $Q = Q_0 \setminus < (i, j), w >$.

Figure 1: The QUEUE Module

procedure ARCCONS(**in** i, j, **out** Δ)
Pre: $(i, j) \in arc(G)$.
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: $(i, j) \in arc(G)$ **and** $w \notin D_j$.
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

3.1 Operations on Queues

The operations we need are described in Figure 1, Procedure INITQUEUB simply initialises 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., po) 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 size of A. 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 < *v,{*A_1*,..,* A_m*}* > where A_k is an arc and v is a value. Procedure ENQUEUE(I, **Δ,** V) adds an element < *v,* {A_1..., A_m} > to the queue, where the A_k are arcs of the form *(j,i),* for each v $\in$ **Δ.** Procedure DEQUEUE picks up an element < w,>, {A_1..., A_m} > 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 con-

```
Algorithm AC-5
Post: let P_0 = D_{i_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 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.
procedure Dequeue(inout Q, out i, j, w)
Post: Status[(i,j),w] = rejected.
```

Figure 4: The Queue Module on Structure Status

straints will be given in the next sect ions. They can also be specialized to produce AC-3 and AC-4 from AC-5,

Procedure ARcCONs(i, j, A) computes the set of values A for variable i that are not supported by *Dj.* Procedure LOCALARCCONS(I, J,W, *A) 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 A to be returned. It is sufficient to compute A_1 to guarantee the correctness of AC-5, However the procedure gives the opportunity to achieve more pruning (up to A_2) while still preserving the soundness of the algorithm. Interestingly enough, the ability to achieve more pruning turns out to be fundamental, for some classes of constraints (e.g. monotonic constraints), in producing an *0(ed)* algorithm.

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(A, *D)* removes the set of values A from *D.* 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 [10] that it generalises. 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.

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

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 size of Δ on each iteration at lines 12, then $s_1 + \ldots + s_p \leq O(ed)$.

Proof

Property 1 holds initially. Assuming that it holds in line 2, it also holds after an iteration of lines 4 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 above theorem can be used to deduce the overall complexity of AC-5 from the complexity of Procedures ArcCons and LocalArcCons. In particular, in AC-3 and AC-4, Procedure ArcCons is necessarily $O(d^2)$ which implies that the overall complexity is at least $O(ed^2)$ since lines 4 to 6 are executed e times. 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 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 next two sections, we characterize two 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 resulting in an AC-5 algorithm for these classes, running in time $O(ed)$. In these sections, we assume a number of primitive operations 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*

```
procedure RemoveElem(in v, inout D)
Post: D = D_0 \ {v}.
function Member(in v, D): Boolean
Post: Member ⇔ (v ∈ D).
function Min(in D): Value
Post: Min = min{v ∈ D}.
function Max(in D): Value
Post: Max = max{v ∈ D}.
function Succ(in v, D): Value
Post: Succ = min{v' ∈ D | v' > v}.
function Pred(in v, D): Value
Post: Pred = max{v' ∈ D | v' < v}
```

Figure 5: The Domain module

```
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 6: ArcCons for Functional Constraints

for reasons that will become clear later.[3] The additional sophistication is necessary to achieve the bound *0(ed)* for monotonic constraints. These primitive operations are assumed to take constant time.

4 Functional Constraints

Definition 5 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.

Convention 6 If *C* is a functional constraint, we denote by *fc(v)* the value *w* such that *C(v, w)* and f_C^{-1} *(w)* the value *v* such that *C[v, w).* Since AC-5 works on arcs, we associate a function f_{ij} to each arc (i, *j)* in such a way that, for constraint C_{ij}, are (i, j) is assigned f_{cij} and arc (j, i) is assigned $f_{C_{ij}}^{-1}$.

The results presented in the paper assume that it takes constant time to compute the functions *fc* and f_C^{-1} 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 6 and 7.

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 *0(d)* and O(1) for functional constraints. Hence we have an optimal algorithm.

Note that if *D* is made up of several on connected domains with distinct orderings, it is always possible to transform the underlying partial ordering into a total ordering.

```
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 7: LocalArcCons for Functional Constraints

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

Figure 8: ArcCons for Monotonic Constraints

Theorem 7 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.

5 Monotonic Constraints

We now consider another class of constraints in this section. 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 constraint is used to simplify the algorithms but it is not restrictive in nature.

Definition 8 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 9 Since AC-5 is working with arcs, we associate to each arc (i, j) three functions f_{ij}, $first_{ij}$, 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 $f_{ij}(w) = max\{v \mid C(v, w)\}$, $first_{ij}$ = Max, $next_{ij}$ = Pred, $\succ_{ij} = >$ while those for arc (j, i) are $f_{ji}(v) = min\{w \mid C(v, w)\}$, $first_{ji}$ = Min, $next_{ji}$ = Succ, $\succ_{ji} = <$.
Moreover, since Procedures ArcCons and LocalArcCons only use f_{ij}, $first_{ij}$, $next_{ij}$, and $\succ_{ij}$ for arc (i, j),

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

Figure 9: LocalArcCons for Monotonic Constraints

Let $S = \{b, \ldots, B\}$
$D_i = \{v_1, \ldots, v_m\} \subseteq S$ with $v_k < v_{k+1}$ and $m > 0$.
Syntax
$D_i.min$: 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
$D_i.min = v_1$
$D_i.max = v_m$
$D_i.element[v]$ iff $v \in D_i$
$D_i.succ[v_k] = v_{k+1}$ $(1 \leq k < m)$
$D_i.pred[v_{k+1}] = v_k$ $(1 \leq k < m)$

Figure 10: DOMAIN of Consecutive Values.

we omit the subscripts in the presentation of the algorithms.

We are now in position to describe the implementation of Procedures ARCCONS and LOCALARCCONS for monotonic constraints. They are depicted in Figures 8 and 9.

Lemma 10 Procedures ARCCONS and LOCALARCCONS fulfill their specifications.

Proof The result for Procedure ARCCONS follows from the monotonicity of the constraint that make sures that the value v can be checked only wrt an extremal value (minimum or maximum depending on the arc).

Procedure LOCALARCCONS computes the set $\Delta = \{v \in D_i \mid v \succ f(first(D_j))\}$. By monotonicity of the constraint, $\Delta \subseteq \Delta_2$, and $\Delta_2 \cap \{v \in D_i \mid v \preceq f(first(D_j))\} = \emptyset$. Hence $\Delta = \Delta_2$ and the postcondition is satisfied. □

Note that, for monotonic constraints, it is more complicated to compute the set Δ_1 as we have no guarantee that the extremal value corresponding to w for variable i is in its domain. If the value is not in the domain, then we have no way to use Procedures PRED and SUCC, and this leads to a non-optimal algorithm.

Procedures ARCCONS is $O(d)$. Procedure LOCALARCCONS has 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 11 Procedure AC-5 is 0(ed) for monotonia constraints wrt *D.*

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

6 Implementation of Domain

In the previous sections, we assume that the primitive operations on domains can be performed in constant time. In this section, we present 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 10. 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 is basically the same but it reasons about indices instead of values and uses an hash-table to test membership to the domain. Although the time complexity of membership is theoretically not 0(1), under reasonable assumption, the expected time to search for an element is O(I).

7 Application

We describe the application of AC-5 to Constraint Logic Programming over finite domains.

Constraint Logic Programming [7] is a class of languageB 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 [19, 17, 18]. 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 (i, x ....). 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 [3] 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 [2, 18].

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 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 12 A *system of constraints* 5 is a pair *(AC, DC)* 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 5.

Definition 13 Let $S = (AC, DC)$ 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 14 Let S be a system of constraints. S is in *solved form* iff any unary constraint $C(X)$ in S is node consistent[3] 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 15 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 16 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 17 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 15 and 16, 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 18 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 19 Let S be a system of constraints. Algorithm 18 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 20 returns *true* if S is satisfiable and *false* otherwise.

Algorithm 20 To check the satisfiability of a system of constraints S: (1) apply Algorithm 18 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 specialisation of AC-5 for basic constraints, we obtain an $O(ed)$ decision procedure.

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

8 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 *0(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 *0(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. Finally, while arc-consistency of functional constraints can also be solved through a reduction to 2-sat [8], it is an open issue to find out if a similar reduction exists for monotonic constraints.

References

[I] R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intttligtnce,* 34:1-38, 1988-

[2] M. Dincbas, H. Simonii, and P. Van Hentenrjck. SoWing Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming,* 8(I-2):76-93, 1990.

[3] M. Dincbas and al The Constraint Logic Programming Language CHIP- In *FGCS S8,* Tokyo, 1988.

[4] B.C. Freuder. Synthesizing Constraint Expressions. *CACM,* 21:958-966, 1978.

[5] J. Gaschnig. A Constraint Satisfaction Method for Inference Making. In *Annual Conf. on Circuit System Theory,* 1974.

[6] R,M. Haralick and G.L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence,* 14:263-313, 1980.

[7] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *ICLP-87,* Melbourne 1987.

[8] S. Kasil On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *AI Journal,* 45:275-286, 1990.

[9] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence,* 10(1), 1978.

[10] A.K. Mack worth. Consistency in Networks of Relations. *AI* Journal; B(1):99-118, 19T7.

[11] A.K- Mackworth. *Constraint Satisfaction,* volume Encyclopedia of Artificial Intelligence. Wiley, 1987.

[13] A,K. Mackworth and E.G. Freuder. The Complexity of some Polynomial Network Consistencj Algorithms for Constraint Satisfaction Problem*. *Artificial Intelligence,* 25:65-74, 1985.

[13] R. Mohr and T.C Henderson. Arc and Path Consistency Revisited. *Artificial* /ntelligence, 28:225-233, 1986.

[14] U. Montanari, Networks of Constraints : Fundamental Propertics and Applications to Picture Processing, *Information Science,* 7(2):96-132, 1974.

[15] Montanari, U and Rossi, F. Constraint Relaxation May Be Perfect In *To Appear in Artificial Intelligence.*

[16] B. Nadel, Constraint Satisfaction Algorithms. *Computational Intelligence,* 5(4):288-324, 1989.

[17] P. Van Hentenrjck. A Framework for Consistency Techniques in Logic Programming, In *IJGAI-87,* Milan, 1987.

[18] P. Van Hentenrjck, Constraint *Satisfaction in Logic Programming*- The MIT Press, Cambridge, MA, 1989.

[19] P. Van Hentenrjck and M. Dincbas. Domains in Logic Programming* In *AAAI-86,* Philadelphia, PA, August 1986.

[20] D- Waits. Generating Semantic Descriptions from Drawing* of Scenes with Shadows. Technical Report AI271, MIT, MA, 1973.