BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-92-M4

''On Generic Consistency Algorithms and their Specializations''

by

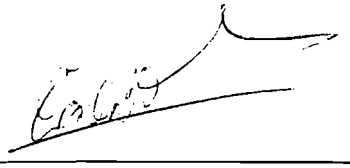
Choh Man Teng

On Generic Consistency Algorithms and their Specializations

Choh Man Teng
Department of Computer Science
Box 1910 Brown University
Providence RI 02912
USA

April, 1992

This thesis by Choh Man Teng is accepted in its present form by the Department of Computer Science as satisfying the thesis requirement for the degree of Master of Science.



Prof. Pascal Van Hentenryck

4/28/92

Date

On Generic Consistency Algorithms and their Specializations

Choh Man Teng
Department of Computer Science
Box 1910 Brown University
Providence RI 02912
USA

April, 1992

Abstract

Techniques for solving constraint satisfaction problems (CSP) are of great interest in many areas, such as artificial intelligence, operations research, and hardware design. Network consistency algorithms aim at reducing the thrashing behavior of backtracking for constraint solvers by considering various local consistency information. The complexity of consistency algorithms typically depends on n, the number of variables, e, the number of arcs, and d, the maximum size of the variable domains. Previous efforts include Mohr and Henderson's [5] arc consistency algorithm AC-4 which is of complexity $O(ed^2)$, and their path consistency algorithm PC-3, together with a corrected version PC-4 of Han and Lee [2], both of which have a complexity of $O(n^3d^3)$.

A generic arc consistency algorithm AC-5 has been proposed by Deville and Van Hentenryck [1]. Algorithm AC-5 is parametrized so that it can be instantiated to $O(ed)$ algorithms for specific classes of constraints. This complexity has been shown for functional constraints and monotonic constraints.

In this paper, we continue the effort on unraveling more classes of constraints which can be specialized to obtain an $O(ed)$ algorithm from the formulation of AC-5. We refine the instantiations for monotonic constraints, and introduce other classes of constraints, namely, anti-functional and piecewise (functional, anti-functional, and monotonic) constraints. An approach similar to that for AC-5 is used to formulate a new generic path consistency algorithm PC-5, and its instantiations for functional and monotonic constraints are investigated. We also devise two generic incremental arc consistency algorithms for hierarchical networks, and present their instantiations for functional, anti-functional and monotonic constraints, with an attempt to generalize binary constraints to multi-variable constraints in the monotonic class.

Contents

1 Introduction

A constraint satisfaction problem (CSP) is a set of constraints relating a finite set of variables over finite domains. Solving a CSP problem involves finding sets comprising of a value from each domain, so that by assigning the values to the corresponding variables, all constraints can be satisfied. Many tasks in artificial intelligence, operations research, and hardware design can be cast as CSP's. The primary technique for solving a CSP is backtracking over different combinations of values for the variables, and the problem is $\mathcal{NP}$-complete in many cases.

Consistency techniques [1, 3, 4, 5, 6, 9, 10] can be used to prune the problem space of a CSP before attempting to find a solution globally. By considering local consistency between variables, we can rule out values which are impossible to be part of any solution and thus reduce the search space for backtracking. Specifically, arc consistency makes sure that the values of each variable are consistent with respect to each individual constraint, while path consistency considers the consistency of pairs of variable labels across multiple constraints. The complexity of arc and path consistency algorithms typically depends on n, the number of variables, e, the number of arcs, and d, the maximum size of the variable domains.

Various arc consistency algorithms have been proposed, and algorithm AC-4 of Mohr and Henderson [5] has been proved to be optimal with a complexity of $O(ed^2)$. While AC-4 is applicable to all binary constraints, the generic arc consistency algorithm AC-5, by Van Hentenryck and Deville [1], parametrizes the processing in a way so that it can take advantage of the properties of specific classes of constraints in the network. By instantiating the parametric procedures in specialized ways, it has been shown that for functional and monotonic constraints, algorithm AC-5 runs in $O(ed)$ time.

Mohr and Henderson also gave a path consistency algorithm PC-3 [5] (later corrected by Han and Lee [2] as PC-4) which has a complexity of $O(n^3d^3)$. It is not clear whether the algorithms are optimal, and the high complexity limits the practicability of path consistency algorithms to small problems only.

This paper is organized as follows. Section 2 defines the basic conventions and notations used in describing a CSP and its associated consistency algorithms. In Section 3 we investigate more classes of constraints which, by instantiating the parametric procedures of algorithm AC-5 according to specific properties of the constraints, can be specialized to give arc consistency algorithms with $O(ed)$ complexity. We refine the instantiations for monotonic constraints, and also show that an $O(ed)$ complexity can be achieved for anti-functional as well as generalized piecewise (functional, anti-functional and monotonic) constraints. In Section 4, we present a new generic path consistency algorithm PC-5 formulated in a way analogous to algorithm AC-5. PC-5 contains two parametric procedures which can be instantiated to give a path consistency algorithm with a complexity lower than that of PC-3. Specializations for functional and monotonic constraints are also given. In Section 5, we generalize binary constraints to multi-variable constraints. Then we construct incremental arc consistency algorithms for binary functional and anti-functional constraints as well as generalized monotonic constraints in hierarchical networks. We show that the algorithms are

optimal for constraints with limited number of variables, such as binary constraints. Lastly, section 6 concludes the discussion.

2 Descriptives

Throughout the paper, we assume that for a given CSP, there are n variables, each taking on a natural number between 1 and n inclusive. A variable i has an associated finite domain D_i, while D is the union of all domains and d is the maximum size of a domain. The constraint between two variables i and j $(i < j)$ is denoted by C_{ij}. We assume that there is at most one constraint between two variables and each constraint relates distinct variables. Only binary constraints are considered in Sections 3 and 4, but we will relax this requirement in Section 5. $C_{ij}(v, w)$ denotes a boolean value which indicates whether the constraint C_{ij} is satisfied when i and j in the constraint are replaced by v and w respectively.

In the graphical notation, a graph G of a CSP contains n nodes, with node i representing variable i. A constraint C_{ij} $(i < j)$ is denoted by two directed arcs, (i, j) and (j, i). Arc (i, j) denotes the constraint C_{ij}, while arc (j, i) denotes C_{ji}, the same constraint in reverse, so that $C_{ij}(v, w)$ implies $C_{ji}(w, v)$ and vice versa for any values v in D_i and w in D_j. The number of edges is denoted by e, which is twice the number of constraints in a binary constraint network. The set of arcs and the set of nodes in a graph G are denoted by $arc(G)$ and $node(G)$ respectively.

In the specification of operations, we take the convention that a subscript 0 to a structure p, that is, p_0, represents the value of p at the start of the call.

3 More Specializations of Algorithm AC-5

In this section, we continue from the results obtained in [1] and present a number of classes of constraints for which the parametric procedures can be instantiated to give $O(ed)$ arc consistency algorithms from AC-5. We also revise the instantiations for monotonic constraints so that a simpler data structure can be used.

3.1 Previous Work

First of all, we give an overview of the previous works on algorithm AC-5. Refer to [1, 9] for the details of the algorithm. In the rest of this section, we assume the domain structure of variables and the queue operations are the same as specified in [9].

Algorithm AC-5 is parametrized in two procedures, ARCCONS and LOCALARCCONS. We reproduce them here in Figures 1 and 2 from [9] as we will be referring to these specifications frequently. Procedure ARCCONS computes the set of values in D_i which are not arc consistent with any value in D_j. Procedure LOCALARCCONS computes the set of values in D_i that becomes unsupported after the value w has been removed from D_j.

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 1: Specification of the Procedures

Algorithm AC-5
Post: let $\mathcal{P}_0 = D_{1_0} \times \ldots \times D_{n_0}$,
$\mathcal{P} = D_1 \times \ldots \times D_n$
G is maximally arc-consistent wrt $\mathcal{P}$ 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

Figure 2: The Arc-consistency Algorithm AC-5

```
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 3: ArcCons for Functional Constraints

It has been shown that if the time complexity of procedure ArcCons is $O(d)$, and that of procedure LocalArcCons is $O(\Delta)$, where Δ is the set of values returned, then the time complexity of algorithm AC-5 is $O(ed)$. Note that we take the convention that $O(\Delta) = O(\max(1, |\Delta|))$. This $O(ed)$ complexity is optimal for subclasses of constraints, and specific instantiations have been given for functional and monotonic constraints, utilizing the following conventions.

Convention 1 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 denotes a value outside the domain for which the constraint holds.

Convention 2 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 $\succ_{ij}$. Given a monotonic constraint C_{ij}, the functions and relation for arc (i, j) are $f_{ij}(w) = max\{v \mid C_{ij}(v, w)\}$, $last_{ij}$ = 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.

For convenience, we also reproduce the instantiations of the two parametric procedures for functional and monotonic constraints in Figures 3–4 and Figures 5–6 respectively.

By utilizing amortized complexity analysis [7], we are able to obtain the same bounds for monotonic constraints without maintaining data structures on the successor and predecessor information of values. We also show that the same complexity can be achieved for other classes of constraints with proper instantiations of ArcCons and LocalArcCons. These subclasses include anti-functional constraints and piecewise (functional, anti-functional, and monotonic) constraints.

3.2 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

```
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 4: LocalArcCons for Functional Constraints

```
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 5: ArcCons for Monotonic Constraints

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

Figure 6: LocalArcCons for Monotonic Constraints

```
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 7: Revised Procedure ArcCons for Monotonic Constraints

```
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 8: Revised Procedure LocalArcCons for Monotonic Constraints

D_i^{init}), thus eliminating the need to update part of the data structure. The revised procedure ArcCons is depicted in Figure 7. 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 Figure 8. 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,[1] but this would prevent the reduction of domains as early as possible.

Theorem 3 With the revised implementation depicted in Figures 7 and 8, Procedure AC-5 is $O(ed)$ for monotonic constraints wrt D.

Proof This proof requires the use of amortized complexity [7] to show that LocalArcCons is $O(d)$ amortized. The number of iterations for a call to the revised version of LocalArcCons is not $O(\Delta)$ 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 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 is thus never considered any more, 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. □

[1] In line 4 in Figure 8, replace $f(last(D_j))$ by $f(w)$

```
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 9: Procedure ARCCONS for Anti-Functional Constraints

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

Figure 10: Procedure LOCALARCCONS for Anti-Functional Constraints

3.3 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 4 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. Procedures ARCCONS and LOCALARCCONS are shown in Figures 9 and 10. 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 $f_{ji}(w)$ 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.[2]

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

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

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: $\Delta = \Delta_0 \cup (S_k^{ij} \cap D_i)$
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 11: The PIECEWISE DECOMPOSITION Module

3.4 Piecewise Constraints

The functional, anti-functional and monotonic constraints 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 6 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 7 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.

Representation of Piecewise Constraints

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

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

Figure 12: PIECEWISE DECOMPOSITION Data Structure

Before presenting the implementation of ARCCONS and LOCALARCCONS for constraints having some particular piecewise decomposition, we show in Figure 11 operations on piecewise decompositions. 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 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:

$S_k^{ij} \cap D_i \neq \emptyset \Rightarrow$ *Status-pd*$[(i,j),k]$ = `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 Figure 12. Its space complexity is $O(d)$ per piecewise decomposition. This data 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 (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.

3.5 Piecewise Functional Constraints

Intuitively, a piecewise functional constraint C_{ij} is a constraint whose domains 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 8 A constraint C_{ij} is *piecewise functional* wrt domains 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 [8] 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 (resp. 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 $C_{ij}(S_k, P_k) \quad (1 \leq k \leq n)$

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 assumes a piecewise decomposition that satisfies PF1–3. The following property states necessary and sufficient conditions for a piecewise functional constraint.

Property 9 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 $\neg C_{ij}(S_k, w) \quad$ (for all $w \in D_j$ and $0 \leq k \leq n$)
(2) $C_{ij}(S_k, w) \Rightarrow \neg C_{ij}(S_{k'}, w) \quad$ (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 that there is 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:

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

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

Figure 13: 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 14: ARCCONS for Piecewise Functional Constraints

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 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 $P_k \neq \emptyset$.

(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 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 Figures 14 and 15. Line 2 handles the group S_0^{ij} containing all the unsupported

```
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 15: 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 16: Procedure ArcCons for Piecewise Anti-Functional Constraints

elements of the initial domain D_i. The procedures use the boolean function Unsupported specified in Figure 13. 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.

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 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 10 Procedure AC-5 is $O(ed)$ for piecewise functional constraints.

3.6 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 (resp. D_j) is not supported by at most one group of D_j (resp. D_i).

Definition 11 A constraint C_{ij} is *anti-functional* wrt D_i, D_j iff $\neg C_{ij}$ is piecewise functional wrt D_i, 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 Figures 16 and 17). 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 12 Algorithm AC-5 is $O(ed)$ for piecewise anti-functional constraints.

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

Figure 17: Procedure LocalArcCons for Piecewise Anti-Functional Constraints

```
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 18: Procedure ArcCons for Piecewise Monotonic Constraints

3.7 Piecewise Monotonic Constraints

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

Definition 13 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 14 As for monotonic constraints, 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: $f_{ij}(k) = max\{\{-1\} \cup \{k' \mid C_{ij}(S_k^{ij}, S_{k'}^{ji})\}\}$, $last_{ij}(a, b) =$ 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 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 Figures 18 and 19. 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.

```
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
```

Figure 19: Procedure LocalArcCons for Piecewise Monotonic Constraints

Theorem 15 Algorithm AC-5 is $O(ed)$ for piecewise monotonic constraints.

4 A Generic Path Consistency Algorithm

In this section we present a new generic path consistency algorithm PC-5 which can be parametrized in a way similar to that of the arc consistency algorithm AC-5. The time complexity of PC-5 depends on the time complexity of the two procedures PathCons and LocalPathCons. It can be instantiated to give algorithm PC-2 [3] of complexity $O(n^3d^5)$ and also algorithm PC-4 [2] of complexity $O(n^3d^3)$. We show that the parametric procedures can be specialized to give path consistency algorithms of a lower complexity for functional and monotonic constraints.

4.1 Constraint Tuples

We start by giving an alternate definition of constraints.

Definition 16 A *constraint tuple over* D is a tuple $\langle v, w\rangle$, where $v, w \in D$.

Definition 17 A *constraint* C_{ij} *over* D_i, D_j is a set of constraint tuples such that $\forall v \in D_i, w \in D_j$, $\langle v, w\rangle \in C_{ij}$ iff C_{ij} is satisfied when i and j in the constraint are replaced by v and w respectively.

The pairs of values admissible for a constraint C are represented as tuples in C. Here we can draw a parallel between path and arc consistency algorithms. A path consistency algorithm removes path inconsistent tuples from the constraints and returns a subset of tuples for each constraint. Similarly, an arc consistency algorithm removes arc inconsistent

values from the domains of variables and returns a subset of values for each domain as a result. We will use this definition of constraints with path consistency in this section.

4.2 Preliminaries

Before presenting the generic algorithm, let us first review some of the basics in the area of path consistency.

Definition 18 Let $(i_0, i_1), \ldots, (i_{m-1}, i_m) \in arc(G)$ and $v_{i_0} \in D_{i_0}, v_{i_m} \in D_{i_m}$. v_{i_0} is *path consistent with* v_{i_m} *wrt the path* $(i_0, \ldots, i_m)$ *in* $D_{i_0}, \ldots, D_{i_m}$ iff $\langle v_{i_0}, v_{i_m} \rangle \in C_{i_0 i_m}$, and $\exists v_{i_1}, \ldots, v_{i_{m-1}}$ such that

1. $v_{i_1} \in D_{i_1}, \ldots, v_{i_{m-1}} \in D_{i_{m-1}}$.
2. $\langle v_{i_0}, v_{i_1} \rangle \in C_{i_0 i_1}, \ldots, \langle v_{i_{m-1}}, v_{i_m} \rangle \in C_{i_{m-1} i_m}$.

Definition 19 Let $i_0, i_m \in node(G)$ and $v_{i_0} \in D_{i_0}$, $v_{i_m} \in D_{i_m}$. v_{i_0} is *path consistent with* v_{i_m} *wrt* D_{i_0}, D_{i_m} iff v_{i_0} is path consistent with v_{i_m} wrt all paths $(i_0, \ldots, i_m)$. We also say that v_{i_0} is *path consistent with node* i_m iff $\exists v_{i_m} \in D_{i_m}$ such that v_{i_0} is path consistent with v_{i_m}.

Definition 20 Let $(i_0, i_1), \ldots, (i_{m-1}, i_m) \in arc(G)$. A path of length m $(i_0, \ldots, i_m)$ is *path consistent wrt* $D_{i_0}, \ldots, D_{i_m}$ iff $\forall v_{i_0} \in D_{i_0}, \exists v_{i_m} \in D_{i_m}$ such that v_{i_0} is path consistent with v_{i_m} wrt $(i_0, \ldots, i_m)$ and vice versa.

Definition 21 Let $\mathcal{P}$ be $D_0 \times \ldots \times D_n$. A graph G is *path consistent wrt* $\mathcal{P}$ iff for all paths $(i_0, \ldots, i_m), m > 0$ and $(i_0, i_1), \ldots, (i_{m-1}, i_m) \in arc(G)$, $(i_0, \ldots, i_m)$ is path consistent wrt $D_{i_0}, \ldots, D_{i_m}$.

Definition 22 Let $\mathcal{P}$ be $D_0 \times \ldots \times D_n$. Let $\mathcal{P}'$ be $D'_0 \times \ldots \times D'_n$ and $\mathcal{P}' \subseteq \mathcal{P}$. G is *maximally path consistent wrt* P' *in* $\mathcal{P}$ iff G is path consistent wrt $\mathcal{P}'$ and there is no other $\mathcal{P}''$ with $\mathcal{P}' \subset \mathcal{P}'' \subseteq \mathcal{P}$ such that G is path consistent wrt $\mathcal{P}''$.

The purpose of a path consistency algorithm is to compute, given a graph G and a set P, a set P' such that G is maximally path consistent wrt P' in P.

Theorem 23 Let $\mathcal{P}$ be $D_0 \times \ldots \times D_n$. A complete graph G is path consistent wrt $\mathcal{P}$ if $\forall i_0, i_1, i_2 \in node(G), (i_0, i_1, i_2)$ is path consistent wrt $D_{i_0}, D_{i_1}, D_{i_2}$.

Theorem 23, due to Montanari [6], states that a complete graph is path consistent if all paths of length two are path consistent. The proof could be constructed easily by an induction on the length of paths in the graph.

Definition 24 Let $\mathcal{P}$ be $D_0 \times \ldots \times D_n$. Two graphs G and G' are *equivalent* iff a path consistency algorithm gives the same solution set for G and G' wrt $\mathcal{P}$.

Corollary 25 Let $\mathcal{P}$ be $D_0 \times \ldots \times D_n$. A graph G is *path consistent wrt* $\mathcal{P}$ iff G can be transformed into an equivalent complete graph G' and $\forall\, i_0, i_1, i_2 \in node(G') : (i_0, i_1, i_2)$ is path consistent wrt $D_{i_0}, D_{i_1}, D_{i_2}$ in G'.

Since Theorem 23 holds for complete graphs only, the above will be the operational definition for our path consistency algorithm. There are various ways of completing a graph, as we will see in the following sections.

4.3 Basic Operations

We introduce here some primitive operations and structures relating to constraints, domains, and the queue for use with algorithm PC-5.

Constraints

We express domain constraints as single variable constraints such that for each value v in the domain of node i, we have a tuple $\langle v, v\rangle$ in C_{ii}. This allows the algorithm to be uniform and simplifies the presentation. Note that we can derive D_i from C_{ii} since $v \in D_i$ implies $\langle v, v\rangle \in C_{ii}$ and vice versa.

We denote by C_{ij}^{init} the original set of constraint tuples between nodes i and j, and by D_i^{init} the given set of domain values of node i. The output of a path consistency algorithm can be defined in terms of the domain constraints. Specifically, given a graph G, the solution of a path consistency algorithm is a set of constraints $C_{ii} \subseteq C_{ii}^{init}$ for each node in $node(G)$ such that G is maximally path consistent wrt $D'_0, \ldots, D'_n$, where $D'_i = \{v \mid \langle v, v\rangle \in C_{ii}\}$.

The operations on constraints are depicted in Figure 20. Function MEMBER checks whether a tuple is in a constraint. SUPPORT returns a set of values w with a tuple $\langle v, w\rangle$ in C_{ij}, while MIN and MAX returns the minimum and maximum values of this set, and SIZE gives its size. Procedure PRUNEELEM removes a tuple $\langle v, w\rangle$ from C_{ij}. We modify a copy of C_{ij}^{init} to give the current C_{ij} so that we can always have access to the initial tuples. We assume that all these basic operations take constant time.

A data stucture supporting such operations is shown in Figure 21, assuming that D consists of consecutive integer values. For a constraint C_{ij}, the fields *element*, *support*, *min*, *max* and *size* corresponds to the functions MEMBER, SUPPORT, MIN, MAX and SIZE respectively. Procedure PRUNEELEM updates the fields in constant time when a tuple $\langle v, w\rangle$ is removed from C_{ij}. This assumes that we can access the element w in $C_{ij}.support[v]$ directly from the pair v–w, which can easily be achieved by keeping a link from $C_{ij}.element[v][w]$ to the corresponding set element. The space complexity of the data structure is $O(d^2)$ for each constraint.

The Queue

The queue operations required for PC-5 are shown in Figure 22. INITQUEUE and EMPTYQUEUE both take constant time. For each tuple in Δ, procedure ENQUEUE puts on the

function MEMBER(**in** $\langle v, w\rangle, D_i, D_j$): boolean
Post: MEMBER $\Leftrightarrow (\langle v, w\rangle \in C_{ij})$.

function SUPPORT(**in** v, C_{ij}): set of values
Post: SUPPORT $= \{w \mid \langle v, w\rangle \in C_{ij}\}$.

function MIN(**in** v, C_{ij}): value
Post: MIN $= \min\{w \mid \langle v, w\rangle \in C_{ij}\}$.

function MAX(**in** v, C_{ij}): value
Post: MAX $= \max\{w \mid \langle v, w\rangle \in C_{ij}\}$.

function SIZE(**in** v, C_{ij}): set of values
Post: SIZE $=\mid$ SUPPORT$(v, C_{ij}) \mid$.

procedure PRUNEELEM(**in** $\langle v, w\rangle, C_{ij}$)
Post: $C_{ij} = C_{ij_0} \setminus \{\langle v, w\rangle\}$.

Figure 20: The CONSTRAINT module for PC-5

Let $D = \{b, \ldots, B\}$.
Let $C_{ij} = \{\langle v_1, v_1\rangle, \ldots, \langle v_m, v_m\rangle\}$ if $i = j$, where $v_1, \ldots, v_m \in D$, $v_k < v_{k+1}$ and $m > 0$.
$= \{\langle v'_1, w'_1\rangle, \ldots, \langle v'_m, w'_m\rangle\}$ if $i \neq j$, where $v'_k \in D_i, w'_k \in D_j$.

Syntax
$C_{ij}.element$: array $[b..B][b..B]$ of boolean.
$C_{ij}.support$: array $[b..B]$ of set of integers $\subseteq D$.
$C_{ij}.min$: array $[b..B]$ of integer.
$C_{ij}.max$: array $[b..B]$ of integer.
$C_{ij}.size$: array $[b..B]$ of integer.

Semantics
$C_{ij}.element[v][w] \Leftrightarrow \langle v, w\rangle \in C_{ij}$.
$C_{ij}.support[v] = \{w_1, \ldots, w_p\}$, where $\langle v, w_k\rangle \in C_{ij}, w_k < w_{k+1}$ for $1 \leq k \leq p$.
$C_{ij}.min[v] = w_1$.
$C_{ij}.max[v] = w_p$.
$C_{ij}.size[v] = p$.

Figure 21: The CONSTRAINT Data Structure

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

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

procedure DEQUEUE(**inout** Q, **out** $i, k, j, \langle v, u\rangle$)
Post: $\langle i, k, j, \langle v, u\rangle\rangle \in Q_0$ and $Q = Q_0 \setminus \{\langle i, k, j, \langle v, u\rangle\rangle\}$.

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

Figure 22: The QUEUE Module for PC-5

queue all length-two paths involving either (i, j) or (j, i) as the first arc. Procedure DEQUEUE returns an element from the queue. We require that procedure ENQUEUE takes $O(s)$ time, where s is the size of Δ, and procedure DEQUEUE takes constant time.

To achieve the bounds, the queue elements can be grouped together as tuples of the form $\langle i, j, E, \langle v, w\rangle\rangle$ for each $\langle v, w\rangle$ in Δ, and E is initially $node(G)$. Procedure DEQUEUE inspects an element $\langle i, j, E, \langle v, w\rangle\rangle$ in the queue, detaches a node k from E, and returns $\langle i, j, k, \langle v, w\rangle\rangle$. We assume that an element $\langle i, j, E, \langle v, w\rangle\rangle$ is removed from the queue when E becomes empty. Using this organization, procedure ENQUEUE takes $O(s)$ time and all other operations take constant time.

4.4 A Generic Path Consistency Algorithm

Now, we are ready to present the generic path consistency algorithm PC-5 with two parametric procedures, the implementations of which are left open.

Parametric Procedures

First of all, let us define extensions of constraints and constraint graphs by the queue Q.

Definition 26 Let Q be the queue containing elements to be processed and Q_{ij} denote the set of tuples $\{\langle v, w\rangle \mid \langle i, j, k, \langle v, w\rangle\rangle \in Q\}$. A constraint $C_{ij/Q}$ is an *extended constraint from C_{ij} and Q* such that $C_{ij/Q} = C_{ij} \cup Q_{ij}$. A graph $G_{/Q}$ is an *extended graph from G and Q* where each constraint C_{ij} in G is extended to $C_{ij/Q}$.

The extended constraint graph takes the original constraint graph and augments each constraint with the tuples that have been removed but are pending in the queue.

The specifications of two parametric procedures, PATHCONS and LOCALPATHCONS, for the generic path consistency algorithm PC-5 are depicted in Figure 23. Procedure PATHCONS computes the set Δ of tuples in C_{ij} that are not path consistent wrt the path (i, k, j).

Let $PC_{ikj}(G, v, w) = \exists u \in D_k : \langle v, u \rangle \in C_{ik}$ and $\langle u, w \rangle \in C_{kj}$.

procedure PATHCONS(**in** i, k, j, **out** Δ)
Pre: $i, k, j \in node(G)$.
Post: $\Delta = \{\langle v, w \rangle \in C_{ij} \mid \neg PC_{ikj}(G, v, w)\}$.

procedure LOCALPATHCONS(**in** $i, k, j, \langle v, u \rangle$, **out** Δ)
Pre: $i, k, j \in node(G)$, and $\langle v, u \rangle \notin C_{ik}$.
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_1 = \{\langle v, w \rangle \in C_{ij} \mid \neg PC_{ikj}(G_{/Q}, v, w)\}$,
$\Delta_2 = \{\langle v', w' \rangle \in C_{ij} \mid \neg PC_{ikj}(G, v', w')\}$.

Figure 23: Specifications of Parametric Procedures

Procedure LOCALPATHCONS returns in Δ a set of tuples of C_{ij} that are path inconsistent wrt (i, k, j) after the tuple $\langle v, u \rangle$ has been removed from the constraint C_{ik}.

The size of Δ computed by procedure LOCALPATHCONS can vary. Δ_1 contains the tuples in C_{ij} that become path inconsistent wrt (i, k, j) due to the removal of the tuple $\langle v, u \rangle$ from C_{ik}. Specifically, we can construct an extended constraint graph $G_{/Q}$ from G, and a tuple $\langle v, w \rangle$ is included in Δ_1 if v and w are not path consistent wrt (i, k, j) in $G_{/Q}$. It is the minimal set required by the path consistency algorithm. In some cases, it is possible, but not always desirable, to prune a larger set of tuples. On the extreme, Δ_2 prunes all tuples in C_{ij} that are path inconsistent wrt (i, k, j) at the point of calling, regardless of whether they can be supported by $\langle v, u \rangle$. The specification of procedure LOCALPATHCONS takes advantage of this fact and allows for both flexibility and efficiency.

Algorithm PC-5

The algorithm for the generic path consistency algorithm PC-5 is depicted in Figure 24. Here procedure PRUNE(Δ, C_{ij}) makes use of procedure PRUNEELEM to remove from C_{ij} each tuple $\langle v, w \rangle$ in Δ.

The structure of the algorithm basically mimics that of algorithm AC-5. In the loop on lines 2–7, procedure PATHCONS identifies the path inconsistent tuples with respect to each path of length two. Note that by considering paths of the form (i, k, i), values in node i that are not path (and arc) consistent with any value in node k are removed from C_{ii}, and hence from D_i. The inconsistent tuples are queued up and processed in the second loop, on lines 8–14, where procedure LOCALPATHCONS is used to prune tuples of C_{ij} which become inconsistent after the removal of a tuple from C_{ik}. For ease of specification, we assume that the algorithm stops as soon as any domain becomes empty, since we can conclude that the graph is not path consistent at that point.

Algorithm PC-5
Post: let $\mathcal{P}_0 = D_{1_0} \times \ldots \times D_{n_0}$,
$\mathcal{P} = D_1 \times \ldots \times D_n$
G is maximally path consistent wrt $\mathcal{P}$ in $\mathcal{P}_0$.

```
begin
1    INITQUEUE(Q);
2    for all i,k,j ∈ node(G) do
3    begin
4        PATHCONS(i,k,j,Δ);
5        ENQUEUE(i,j,Δ,Q);
6        PRUNE(Δ,C_ij)
7    end;
8    while not EMPTYQUEUE(Q) do
9    begin
10       DEQUEUE(Q,i,k,j,⟨v,u⟩);
11       LOCALPATHCONS(i,k,j,⟨v,u⟩,Δ);
12       ENQUEUE(i,j,Δ,Q);
13       PRUNE(Δ,C_ij)
14   end
end
```

Figure 24: The Path Consistency Algorithm PC-5

Correctness

The correctness of algorithm PC-5 follows from the correctness of algorithm PC-4 [2]. Algorithm PC-4 is the baseline case of algorithm PC-5, where the implementation of LOCALPATHCONS does not use node j but relies on explicit data structures to determine which paths and tuples are to be reconsidered, and Δ_1 is computed in every call. Obviously, a larger Δ of PC-5 would only increase the efficiency but not affect the correctness of the algorithm.

Note that algorithm PC-2 [3] can also be derived from PC-5 by specific instantiations of the parametric procedures. The implementation of algorithm PC-2 does not make use of the tuple $\langle v, u \rangle$ in LOCALPATHCONS, but computes Δ_2 every time.

Complexity Bounds

We associate a data structure $Status[i, k, v, u, j]$ with each $i, k, j \in node(G), v, u \in D$. The semantics of $Status[i, k, v, u, j]$ is as follows.

$$
\begin{aligned}
Status[i,k,v,u,j] &= \texttt{present} \quad \text{iff } \langle v,u \rangle \in C_{ik}, \\
&= \texttt{suspended} \text{ iff } \langle v,u \rangle \notin C_{ik} \;\&\; \langle i,k,j,\langle v,u \rangle \rangle \in Q, \\
&= \texttt{rejected} \quad \text{iff } \langle v,u \rangle \notin C_{ik} \;\&\; \langle i,k,j,\langle v,u \rangle \rangle \notin Q.
\end{aligned}
$$

procedure INITQUEUE(**out** Q)
Post: $\forall i,k,j \in node(G), \forall v,u \in D : Status[i,k,v,u,j]$ = **present** if $\langle v,u\rangle \in C_{ik}$,
= **rejected** otherwise.

function EMPTYQUEUE(**in** Q)
Post: EMPTYQUEUE = $\forall i,k,j \in node(G), \forall v,u \in D : Status[i,k,v,u,j] \neq$ **suspended**.

procedure DEQUEUE(**inout** Q, **out** $i,k,j,\langle v,u\rangle$)
Pre: $Status[i,k,v,u,j]$ = **suspended**.
Post: $Status[i,k,v,u,j]$ = **rejected**.

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

Figure 25: Manipulations of Structure *Status*

The data structure is manipulated according to Figure 25. We now proceed to prove some properties of algorithm PC-5.

Property 27 Algorithm PC-5 has the following properties.

1. PC-5 preserves the semantics of *Status* on lines 2 and 8.
2. PC-5 enqueues and dequeues at most $O(n^3d^2)$ elements.
3. If $s_1,\ldots,s_p$ are the numbers of new elements in Q after successive iterations of line 12, then $s_1 + \ldots + s_p \leq O(n^3d^2)$.

Proof

1. Property 1 holds initially after INITQUEUE. Line 5 changes the status of the elements $\langle i,j,v,w,k\rangle$ for each $\langle v,w\rangle$ in Δ from **present** to **suspended** and line 6 removes the tuples in Δ from C_{ij} immediately after they are put onto the queue. Thus, property 1 holds on line 2 and still holds when the execution first reaches line 8. Line 10 changes the status of the element dequeued from Q from **suspended** to **rejected**. Line 12 and 13 have similar effects on *Status* as lines 5 and 6. Thus, property 1 holds on line 8 also.
2. Each element in *Status* can assume each of the three values **present**, **suspended**, and **rejected** at most once. Thus, there can be at most $O(n^3d^2)$ enqueues, where the status of an element changes from **present** to **suspended**, and at most $O(n^3d^2)$ dequeues, where the status of an element changes from **suspended** to **rejected**.

3. Property 3 follows from Property 2 and the preconditions of procedure ENQUEUE. An element is not enqueued again if it is already in the queue.

□

Theorem 28 Given a time complexity of $O(d^2)$ for procedure PATHCONS and a time complexity of $O(\Delta)$ for procedure LOCALPATHCONS, algorithm PC-5 is bounded by $O(n^3d^2)$.

Note that if the complexity of PATHCONS is $O(t)$, the loop at lines 2–7 takes $O(n^3) \cdot O(t)$ time. Also, if LOCALPATHCONS takes $O(\Delta)$ time, the loop at lines 8–14 has a complexity of $O(q)$, where q is the total number of elements that can be enqueued throughout the execution of PC-5. These observations will become helpful when we consider functional constraints.

4.5 Functional Constraints

Let us consider path consistency for functional constraints. We can formulate the definition of a functional constraint in terms of constraint tuples.

Definition 29 A constraint C is a *functional constraint wrt* D iff $\forall v$ (resp. w) $\in D$, there exists at most one value w (resp. v) $\in D$ such that $\langle v, w\rangle \in C$.

Convention 30 We denote by $f_{ij}(v)$ the value w, if it exists, where $w \in D_j^{init}$ and $\langle v, w\rangle \in C_{ij}$. The function returns a non-existent value outside D_j^{init} if there does not exist a tuple $\langle v, w\rangle$ in C_{ij}, or if v is outside the domain D_i^{init}.

Theorem 31 There is an implicit functional constraint relating two nodes i and j if there is a path between i and j which consists of only functional constraints.

Suppose there is no arc (i, j) but there is a path $(i, k_0, \ldots, k_m, j)$ between i and j where $C_{ik_0}, \ldots, C_{k_mj}$ are all functional constraints. Suppose there exists a value v in D_i such that v is consistent with two distinct values w and w' in D_j. Then, we have two distinct values u_m and u'_m in D_{k_m} such that $\langle u_m, w\rangle, \langle u'_m, w'\rangle \in C_{k_mj}$ (since C_{k_mj} is functional). Unrolling through $k_{m-1}, \ldots, k_0$, we have $u_0, u'_0 \in D_{k_0}, u_0 \neq u'_0$ and both $\langle v, u_0\rangle$ and $\langle v, u'_0\rangle \in C_{ik_0}$, which contradicts the defintion of functional constraints.

Representation

We can devise a strategy for completing a connected functional graph in accordance to the implicit functional constraints between the nodes. This can be easily done in $O(n^2d)$ time by considering pairs of nodes in increasing order of the distance (path length) between them. If there is no arc between i and j, we can assign to C_{ij} the set $\{\langle v, w\rangle \mid v \in D_i, w \in D_j,$ and $w = f_{kj}(f_{ik}(v))\}$ where (i, k, j) is the first encountered functional path. Before any pair of nodes i and j which are connected by an $(m+1)$-length path $(i, \ldots, k, j)$ is considered,

```
procedure PATHCONS(in i, k, j, out Δ)
  begin
1   Δ := ∅;
2   ∀v ∈ D_i^init do
3   begin
4     w := f_ij(v);
5     u := f_ik(v);
6     if u ∉ D_k and ⟨v, w⟩ ∈ C_ij then Δ := Δ ∪ {⟨v, w⟩}
7     else
8     begin
9       w' := f_kj(u);
10      if w ≠ w' and ⟨v, w⟩ ∈ C_ij then Δ := Δ ∪ {⟨v, w⟩}
11    end
12  end
  end
```

Figure 26: Procedure PATHCONS for Functional Constraints

all arcs for nodes of distance m apart must have been completed, and thus arc (i, j) can be made functional by considering (i, k), which is of path length at most m, and (k, j), which is of path length one.

We only need to consider a complete functional graph in which there is a functional arc between each pair of nodes. If the graph is not complete to start with, we can derive an equivalent complete subgraph for each connected component in the graph by using the preprocessing technique described above. Since unconnected parts cannot interfere with each other, we can then apply the path consistency algorithm on each of the connected complete subgraphs and then return as the solution the union of the results obtained from each of the subproblems.

Instantiation of Parametric Procedures

The instantiations of procedures PATHCONS and LOCALPATHCONS for functional constraints are given in Figures 26 and 27 respectively. Only one value needs to be checked on each arc, since the constraints are functional. In procedure LOCALPATHCONS, we can remove $\langle v, f_{ij}(v)\rangle$ immediately after $\langle v, u\rangle$ is removed from C_{ik} because we are sure that the only value, if any, in D_k^{init} that is consistent with $\langle v, f_{ij}(v)\rangle$ is u.

Complexity Analysis

It is clear that the two procedures satisfy their specifications, with LOCALPATHCONS computing Δ_1. The complexity of PATHCONS is $O(d)$ while that of LOCALPATHCONS is $O(1)$. Note that the number of enqueues is bounded by $O(n^3 d)$, since for a functional constraint,

```
procedure LocalPathCons(in i, k, j, ⟨v, u⟩, out Δ)
   begin
1     if ⟨v, f_ij(v)⟩ ∈ C_ij then Δ := Δ ∪ {⟨v, f_ij(v)⟩}
2     else Δ := ∅
   end
```

Figure 27: Procedure LocalPathCons for Functional Constraints

there can be at most one tuple involving v in C_{ij} for each $v \in D_i^{init}$. Therefore, LocalPathCons is bounded by $O(n^3d)$ globally.

Theorem 32 Algorithm PC-5 is of complexity $O(n^3d)$ for functional constraints.

4.6 Monotonic Constraints

Now let us consider path consistency of monotonic constraints. Again we first define monotonic constraints in terms of constraint tuples.

Definition 33 A constraint C is *monotonic wrt* D iff there exists a total ordering on D such that, for any value v, w in D, $\langle v, w\rangle \in C$ implies that $\langle v', w'\rangle \in C$ for all values $v', w' \in D$ and $v' \leq v, w' \geq w$.

We also make the following observations.

Property 34 Monotonicity of arcs does not imply monotonicity of paths.

That is, given a monotonic constraint C_{ij}, we cannot guarantee that v' is path consistent with w' even if v is path consistent with w, where $v' \leq v$ and $w' \geq w$.

A counter example suffices to prove this property. Given the constraints

$$\begin{aligned} C_{ij}: &\quad i \leq j, \\ C_{ik}: &\quad i \geq k, \\ C_{kj}: &\quad k \geq j, \end{aligned}$$

each v in D_i is consistent with at most one value of node j wrt the path (i, k, j), namely, v itself, which does not comply with monotonicity. However, we can derive a weaker property for path consistency.

Property 35 Let $S_{i,\ldots,j}(v)$ and $S_{i,\ldots,j}(v')$ be the sets of values in D_j which are path consistent with v and v' of D_i, $v > v'$, respectively wrt the path $(i, \ldots, j)$. Then, we have $\max(S_{i,\ldots,j}(v)) \geq \max(S_{i,\ldots,j}(v'))$ and $\min(S_{i,\ldots,j}(v)) \geq \min(S_{i,\ldots,j}(v'))$.

The proof can be formulated as an induction of the path length, starting by observing that this property holds initially for a path of length one, which is the monotonic constraint itself. Now, assume that the property holds for all paths of length n or less, $n \geq 1$. A length-$(n+1)$ path $(i, \ldots, k, j)$ is the concatenation of a length-n path $(i, \ldots, k)$ and a length-one path (k, j). Suppose $\exists v, v' \in D_i, v > v'$ and $w, w' \in D_j, w < w'$, where $w = \max(S_{i,\ldots,k,j}(v))$, $w' = \max(S_{i,\ldots,k,j}(v'))$. Then, we can infer that for any $u' \in D_k$ supporting $\langle v', w' \rangle$, $\langle v, u' \rangle \notin C_{ik}$. Thus, we have the contradicting conclusion that $\max(S_{i,\ldots,k}(v)) < \max(S_{i,\ldots,k}(v'))$, since if $\max(S_{i,\ldots,k}(v)) \geq \max(S_{i,\ldots,k}(v'))$, u' will be in $S_{i,\ldots,k}(v)$. A similar argument can be applied to the minimum values. Hence by induction, the property holds for all paths.

Property 36 Given two consecutive values v and v' in D_i, $v > v'$, and their respective sets of support $S_{i,\ldots,j}(v)$ and $S_{i,\ldots,j}(v')$ in D_j wrt the path $(i, \ldots, j)$, a value $w \in D_j$ is not consistent with any value in D_i wrt $(i, \ldots, j)$ if $\max(S_{i,\ldots,j}(v')) < w < \min(S_{i,\ldots,j}(v))$.

Corollary 37 Any value $w \in D_j$ which is not consistent with $v \in D_i$ wrt a path (i, k, j) falls into one of the following ranges.

1. $w < \min(S_{kj}(u_{min}))$ where $u_{min} = \min(S_{ik}(v))$.
2. $\max(S_{kj}(u')) < w < \min(S_{kj}(u))$ where $u', u \in S_{ik}(v), u' < u$ and $\nexists u'' \in S_{ik}(v)$ such that $u' < u'' < u$.
3. $w > \max(S_{kj}(u_{max}))$ where $u_{max} = \max(S_{ik}(v))$.

This is trivially true since if w is not consistent with v wrt (i, k, j), then $w \notin S_{kj}(u)$ for any $u \in S_{ik}(v)$. The above three ranges constitute the complementary set of values to the set $S = \cup_{u \in S_{ik}(v)} S_{kj}(u)$.

Representation

Now we introduce $TRUE$ constraints to complete a monotonic constraint graph.

Convention 38 We denote by $TRUE_{ij}$ a dummy relation which allows all combinations of values between i and j, i.e., $\forall v \in D_i, \forall w \in D_j : \langle v, w \rangle \in TRUE_{ij}$.

Convention 39 Given a graph G, we can transform G into an equivalent complete graph G' such that C'_{ij} is C_{ij} if $(i, j) \in arc(G)$ and $TRUE_{ij}$ otherwise.

We can insert dummy arcs $TRUE_{ij}$ between every pair of nodes i and j where C_{ij} does not exist already. The graph G' so obtained gives the same arc and path consistency results as graph G, since the $TRUE_{ij}$ relations do not have any effect on the original arcs. From now on, we will take the graph G to be G'.

Property 40 A $TRUE$ constraint is a monotonic constraint.

Let $i, k, j \in node(G)$.
We take the convention that $\min(\emptyset) = +\infty$ and $\max(\emptyset) = -\infty$.
Let $SPC_{ikj}(v, w) = \{u \mid \langle v, u\rangle \in C_{ik} \text{ and } \langle u, w\rangle \in C_{kj}\}$.

function FUNCTIONAL(C_{ij}): boolean
Post: FUNCTIONAL $\Leftrightarrow C_{ij}$ is a functional constraint.

function EMPTYSUPPORT(i, k, j, v, w): boolean
Post: EMPTYSUPPORT $\Leftrightarrow S \cap S' = \emptyset$, where $S = \{u \mid \langle v, u\rangle \in C_{ik}\}$ and $S' = \{u' \mid \langle w, u'\rangle \in C_{jk}\}$.

function PREVSUPPORT(v, i, k, j, u): value
Post: PREVSUPPORT $= \max\{u' \mid u' < u, \exists w : u' \in SPC_{ikj}(v, w)\}$.

function NEXTSUPPORT(v, i, k, j, u): value
Post: NEXTSUPPORT $= \min\{u' \mid u' > u, \exists w : u' \in SPC_{ikj}(v, w)\}$.

function UNSUPPORTED(v, w, w', i, j): set of tuples
Post: UNSUPPORTED $= \{\langle v, x\rangle \in C_{ij} \mid w < x < w'\}$.

Figure 28: The Monotonic Constraint Module for Path Consistency

No special arrangement is needed for incorporating $TRUE$ constraints into a monotonic constraint network. We can express a constraint $C_{ij} = TRUE_{ij}$ as a monotonic constraint where $\langle \max(D_i), \min(D_j)\rangle \in C_{ij}$. All other combinations of values allowable in $TRUE_{ij}$ follow from this tuple with the definition of monotonic constraints. Thus, $TRUE_{ij}$ can be denoted by $\{\langle v, w\rangle \mid v \in D_i, w \in D_j\}$ without violation of monotonicity.

Property 41 Let C_{ii} be a domain constraint. We have (1) C_{ii} is not a monotonic constraint, (2) C_{ii} satisfies property 35, and (3) C_{ii} is a functional constraint.

Domain constraints should not cause a problem since although they are not monotonic, they satisfy the weaker property described earlier. However, to make the algorithm more general, we devise the instantiations of the parametric procedures so that both monotonic and functional constraints can be incorporated at the same time. Note that domain constraints are also functional constraints.

The operations required for monotonic constraints are depicted in Figure 28. Function FUNCTIONAL states whether a given constraint is functional. Function EMPTYSUPPORT checks whether there is any value u in domain k which supports the tuple $\langle v, w\rangle$ in C_{ij} wrt the path (i, k, j). PREVSUPPORT and NEXTSUPPORT return respectively the predecessor and succssor of u in the domain of k which support any tuple involving v in C_{ij} wrt (i, k, j). Function UNSUPPORTED returns the set of tuples $\langle v, x\rangle$ with x in between w and w'.

Function EMPTYSUPPORT can be computed in constant time by comparing the maximum and minimum values of the sets of support. Note that for functional constraints, the support

Let $i, k, j \in node(G)$ and $v, u, w \in D$.
We take the convention that $\min(\emptyset) = +\infty$ and $\max(\emptyset) = -\infty$.
Let $SPC_{ikj}(v, w) = \{u \mid \langle v, u\rangle \in C_{ik}$ and $\langle u, w\rangle \in C_{kj}\}$.

Syntax

$next[i, k, v, u, j]$: value.
$prev[i, k, v, u, j]$: value.
$checked[i, k, v, u, j]$: boolean.

Semantics

$next[i, k, v, u, j] = \min\{u' \mid u' > u$ and $\exists w : u' \in SPC_{ikj}(v, w)\}$.
$prev[i, k, v, u, j] = \max\{u' \mid u' < u$ and $\exists w : u' \in SPC_{ikj}(v, w)\}$.
$checked[i, k, v, u, j] \Rightarrow$ UNSUPPORTED$(v,$ MAX$(n, C_{kj}),$ MIN$(n', C_{kj}), i, j) = \emptyset$, where
$n =$ PREVSUPPORT(v, i, k, j, u), and $n' =$ NEXTSUPPORT(v, i, k, j, u).

Figure 29: Data Structure for Monotonic Constraints

set consists of only one element. We assume that all operations in Figure 28 take constant (amortized) time to execute except function UNSUPPORTED which takes $O(s)$ (amortized) time, where s is the size of the set returned. The bounds can be achieved by employing an $O(n^3d^2)$ data structure as specified in Figure 29. The structures $next[i, k, v, u, j]$ and $prev[i, k, v, u, j]$ return respectively a successor and a predecessor of u such that for any value x between u and u', x is not supporting any tuples $\langle v, w\rangle \in C_{ij}$ wrt the path (i, k, j). Note that *next* contains $+\infty$ and *prev* contains $-\infty$ if there is no value in the domain of D_k^{init} that satisfies the requirements. The element $checked[i, k, v, u, j]$ marks that the tuple $\langle v, u\rangle$ cannot possibly prune any more tuples in C_{ij} wrt (i, k, j), as reflected in the null set returned by UNSUPPORTED. The structure is maintained with *next* and *prev* so that $checked[i, k, v, x, j]$ is set to $TRUE$ for any x where $next[i, k, v, u, j] = u'$ and $u < x < u'$, or $prev[i, k, v, u, j] = u'$ and $u' < x < u$.

Instantiation of the Parametric Procedures

Now we are ready to present the parametric procedures for monotonic constraints. Procedures PATHCONS and LOCALPATHCONS are depicted in Figures 30 and 31 respectively. Note that there is no special treatment for $TRUE$ relations, since $TRUE$ constraints are inherently monotonic constraints. For simplicity, we assume that the functions MAX(v, C_{ij}) and MIN(v, C_{ij}) return $-\infty$ and $+\infty$ respectively when v is not in D_i^{init}.

For each tuple $\langle v, w\rangle$ in C_{ij}, procedure PATHCONS checks whether there is any value in domain k which supports $\langle v, w\rangle$, and removes the tuple if it is unsupported wrt the path (i, k, j). In procedure LOCALPATHCONS, lines 2–3 remove the tuple $\langle v, f_{kj}(u)\rangle$ since the value u can be the only value in k which is consistent with $f_{kj}(u)$ in j wrt the functional constraint C_{kj}. Similarly, lines 4–5 remove all tuples involving v in C_{ij} since v is not consistent with any value in k other than u wrt the functional constraint C_{ik}. Line 6 checks the

```
procedure PathCons(in i, k, j, out Δ)
  begin
1   Δ := ∅;
2   ∀⟨v, w⟩ ∈ C_ij do
3       if EmptySupport(i, k, j, v, w) then Δ := Δ ∪ {⟨v, w⟩}
  end
```

Figure 30: Procedure PathCons for Monotonic Constraints

```
procedure LocalPathCons(in i, k, j, ⟨v, u⟩, out Δ)
  begin
1   Δ := ∅;
2   if Functional(C_kj^init) then
3       Δ := {⟨v, f_kj(u)⟩ | ⟨v, f_kj(u)⟩ ∈ C_ij}
4   else if Functional(C_ik^init) then
5       Δ := {⟨v, w⟩ | ⟨v, w⟩ ∈ C_ij}
6   else if ¬checked[i, k, v, u, j] then
7   begin
8       n := PrevSupport(v, i, k, j, u);
9       n' := NextSupport(v, i, k, j, u);
10      Δ := Unsupported(v, Max(n, C_kj), Min(n', C_kj), i, j)
11  end
  end
```

Figure 31: Procedure LocalPathCons for Monotonic Constraints

structure $checked[i, k, v, u, j]$ where both C_{ik} and C_{kj} are monotonic, and proceeds only if *checked* is not $TRUE$, since we are sure that the Δ computed will be empty for those elements with *checked* being $TRUE$. We compute the values n and n' which are respectively the closest value below and above u supporting any tuple in C_{ij} with v. Line 10 removes all tuples $\langle v, x\rangle$ where x falls in between but is not covered by the support ranges of n and n' in domain j.

Complexity Analysis

The formulation of procedure PathCons is derived directly from the definition of path consistency. The correctness of procedure LocalPathCons follows from the property of functional constraints for the first two cases and from Corollary 37 for monotonic constraints in the last case.

The complexity of procedure PathCons is $O(d^2)$, since there can be at most $O(d^2)$ tuples in a constraint. For procedure LocalPathCons, the first case takes constant time, while

the second takes $O(\Delta)$ time (line 5). For the last case, lines 8–9 can be computed in constant amortized time, and line 10 is of complexity $O(\Delta)$ amortized. Therefore, the procedure has an overall complexity of $O(n^3d^2)$.

Theorem 42 Algorithm PC-5 is of complexity $O(n^3d^2)$ for functional and monotonic constraints combined.

5 Incremental Arc Consistency Algorithms for Hierarchical Networks

Arc consistency can be maintained incrementally in hierarchical constraint networks. Specific data about the constraints are kept to reflect the state of the network. By updating these data during each insertion and deletion of constraints[3], we can derive and maintain arc consistency dynamically.

In this section, we specify two generic incremental arc consistency algorithms for insertion and deletion of constraints. Specializations for functional, anti-functional and monotonic constraints are discussed, with an attempt to relax the binary requirement of constraints to allow for three or more variables in monotonic constraints.

5.1 Generalized Constraints

Until now, we only deal with binary constraints, but we can extend binary constraints to constraints with arbitrary number of variables. First of all, let us fix the notation for expressing multi-variable constraints.

Definition 43 A constraint $C([i_1,\ldots,i_n],[j_1,\ldots,j_m])$ is an $(n+m)$-variable *generalized constraint* between the variables $i_1,\ldots,i_n,j_1,\ldots,j_m$.

Note that a binary constraint can also be expressed in its equivalent generalized form, namely, $C_{ij} = C([i],[j])$.

Convention 44 There are $(n+m)$ edges associated with each $(n+m)$-variable constraint $C([i_1,\ldots,i_n],[j_1,\ldots,j_m])$, each supporting a variable from the set $\{i_1,\ldots,i_n,j_1,\ldots,j_m\}$. We denote by $(x, C([i_1,\ldots,i_n],[j_1,\ldots,j_m]))$ the edge associating variable x with the constraint $C([i_1,\ldots,i_n],[j_1,\ldots,j_m])$, where $x \in \{i_1,\ldots,i_n,j_1,\ldots,j_m\}$.

Convention 45 Let $C([i_1,\ldots,i_n],[j_1,\ldots,j_m])$ be a generalized constraint and c be the edge $(x, C([i_1,\ldots,i_n],[j_1,\ldots,j_m]))$, $x \in \{i_1,\ldots,i_n,j_1,\ldots,j_m\}$. For ease of notation, we will omit the arguments for C and denote c by (x,C) when it is clear from the text which constraint the edge c is associated with.

[3]The algorithms for the insertion and deletion of constraints can easily be extended to similar operations on variables and domain values in the network.

Definition 46 Let $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ be a generalized constraint. Let (k, C) be the edge associating node k and constraint C, where $k \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\}$. Then *trend*$(k, C)$ is *descending* if $k \in \{i_1, \ldots, i_n\}$, and *ascending* if $k \in \{j_1, \ldots, j_m\}$.

Definition 47 We define a relation between the variables in a constraint so that for a constraint $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$, we have

$$\begin{aligned} i_1 < j_1, i_1 < j_2, \ldots, i_1 < j_m, \\ \vdots \\ i_n < j_1, i_n < j_2, \ldots, i_n < j_m. \end{aligned}$$

A hierarchical constraint network is a constraint network in which the above relation imposes a partial ordering on the variables, or, in other words, there does not exist a sequence $x_1 < x_2 < \ldots < x_1$ where x_i are variables in the network.

Intuitively, in the graphical representation, a hierarchical constraint network is a directed graph without cycles. There can be multiple edges or paths between two nodes, but they have to be directed from the same node to the other.

5.2 Basic Structures

Now, we look at the structures of a domain and the queue for use with the incremental arc consistency algorithms.

The Domain

We employ the same data structure for the domains as described in [9]. Note that in order to keep track of the additional data required for updating the arc consistency information, we have to distinguish between the two versions of a domain, D_i^{init} and D_i. In D_i^{init}, we keep the values that have been assigned to the domain of node i regardless of their status, so that values that have been removed by the addition of a constraint can later be reactivated by some deletion operations. The values which are currently "active", or supported, in domain i are represented in D_i. When a value in domain i becomes unsupported, it is removed from D_i, but not from D_i^{init}, while when a new value is introduced into the domain, it is added to D_i^{init} and also D_i if it can be supported under the current state of the constraint network. The space complexity for this structure is $O(d)$ for each domain.

The Queue

The queue operations required are depicted in Figure 32. The operations are similar to those for AC-5 except that the elements in the queue are of the form $\langle (i, C), j, w \rangle$, where (i, C) is an edge supported by j and w is a value that has been removed from D_j. We assume that INITQUEUE, EMPTYQUEUE and DEQUEUE each takes constant time, and procedure ENQUEUE takes $O(\mid \Delta \mid)$ time.

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

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

procedure DEQUEUE(**inout** Q, **out** $(i, C), j, w$)
Pre: $\langle (i, C), j, w \rangle \in Q_0$.
Post: $Q = Q_0 \setminus \{\langle (i, C), j, w \rangle\}$.

procedure ENQUEUE($(j, C), \Delta$, **inout** Q)
Pre: $\Delta \subseteq D_j$.
Post: $Q = Q_0 \cup \{\langle (i, C'), j, w \rangle \mid w \in \Delta$ and $trend(i, C') = trend(j, C), trend(j, C') \neq trend(j, C)\}$.

Figure 32: The Queue Module for Incremental Arc Consistency Algorithms

5.3 The Generic Algorithms

Now we present two generic arc consistency algorithms for inserting and deleting edges incrementally. Each algorithm contains two parametric procedures which can be implemented in different ways for different classes of constraints.

Insertion

The generic incremental arc consistency algorithm INSERTEDGE for edge insertion is shown in Figure 33. Note that the insertion of a constraint $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ corresponds to the insertion of $(n + m)$ edges in the form (k, C) with $k \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\}$. The specifications for the parametric procedures ARCINS, LOCALARCINS, given in Figure 34, make use of extended domains and the extended graph which are defined as follows.

Definition 48 Let Q be the queue containing elements pending to be processed and (i, C) be a generalized constraint. Let $D_{j/Q}(i, C)$ be the *extended domain from* D_j *with* Q *and* (i, C) such that $D_{j/Q}(i, C) = D_j \cup \{w \mid \langle (i, C), j, w \rangle \in Q\}$. Let $G_{/Q}(i, C)$ be the *extended graph from* G *with* Q *and* (i, C) where each domain D_j in G is extended to $D_{j/Q}(i, C)$.

Procedure ARCINS takes a newly inserted edge (i, C) and puts in Δ all values in D_i that should be pruned. Procedure LOCALARCINS returns in Δ all values of D_i which become unsupported after the removal of the value w from D_j. Again, we define Δ in terms of Δ_1 and Δ_2 which are the minimal and maximal sets that can be computed by LOCALARCINS.

Algorithm INSERTEDGE contains two main steps. The first step takes the newly inserted edge (i, C) and checks on line 3 whether it prunes the currently active domain of node i. If so, all edges that are constrained by node i are enqueued on line 4. Lines 6–12 remove an element from the queue and applies LOCALARCINS to it which may put more elements onto

```
Algorithm InsertEdge(in (i, C))
  begin
1    InitQueue(Q);
2    edge(G) := edge(G) ∪ {(i, C)};
3    ArcIns((i, C), Δ);
4    Enqueue((i, C), Δ, Q);
5    Remove(Δ, D_i);
6    while ¬EmptyQueue(Q) do
7    begin
8       Dequeue(Q, (i', C'), j, w);
9       LocalArcIns((i', C'), j, w, Δ);
10      Enqueue((i', C'), Δ, Q)
11      Remove(Δ, D_i');
12   end
  end
```

Figure 33: Algorithm InsertEdge

Let $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ be a generalized constraint.
Let (i, C) be an edge associating node i with the constraint C.
Let $AC(G, (i, C), v) = \exists v_{i_1} \in D_{i_1}, \ldots, v_{i_n} \in D_{i_n}, w_{j_1} \in D_{j_1}, \ldots, w_{j_m} \in D_{j_m}$
such that $v_i = v$ and $C([v_{i_1}, \ldots, v_{i_n}], [w_{j_1}, \ldots, w_{j_m}])$.

procedure ArcIns(**in** (i, C), **out** Δ)
Post: $\Delta = \{v \in D_i \mid \neg AC(G, (i, C), v)\}$.

procedure LocalArcIns(**in** $(i, C), j, w, \Delta$)
Pre: $w \notin D_j$.
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_1 = \{v \in D_i \mid \neg AC(G_{/Q}(i, C), (i, C), v)\}$,
$\Delta_2 = \{v \in D_i \mid \neg AC(G, (i, C), v)\}$.

Figure 34: Specifications of Parametric Procedures for InsertEdge

```
Algorithm DELETEEDGE(in (i, C))
  begin
1    INITQUEUE(Q);
2    edge(G) := edge(G) \ {(i, C)};
3    ARCDEL((i, C), Δ);
4    ENQUEUE((i, C), Δ, Q);
5    INSERT(Δ, D_i);
6    while ¬EMPTYQUEUE(Q) do
7    begin
8        DEQUEUE(Q, (i', C'), j, w);
9        LOCALARCDEL((i', C'), j, w, Δ);
10       ENQUEUE((i', C'), Δ, Q)
11       INSERT(Δ, D_i');
12   end
  end
```

Figure 35: Algorithm DELETEEDGE

the queue. The loop is repeated until the queue is empty. Procedure REMOVE is used on lines 5 and 11 to remove the values in Δ from the corresponding domain.

For simplicity, we assume that an edge would be rejected by the system if its insertion would result in an empty domain. This requires additional back up information which can be easily stored as a stack.

Deletion

The incremental arc consistency algorithm DELETEEDGE for edge deletion is shown in Figure 35 and the specifications of its two parametric procedures are depicted in Figure 36. The notion of contracted domains and a contracted graph is defined below.

Definition 49 Let Q be the queue containing elements pending to be processed and (i, C) be a generalized constraint. Let $D_{j\backslash Q}(i, C)$ be the *contracted domain from* D_j *with* Q *and* (i, C) such that $D_{j\backslash Q}(i, C) = D_j \backslash \{w \mid \langle (i, C), j, w\rangle \in Q\}$. Let $G_{\backslash Q}(i, C)$ be the *contracted graph from* G *with* Q *and* (i, C) where each domain D_j in G is contracted to $D_{j\backslash Q}(i, C)$.

The structure of algorithm DELETEEDGE is similar to that of INSERTEDGE. Procedure ARCDEL takes an edge and returns in Δ all values in D_i^{init} which can be reactivated. Procedure LOCALARCDEL computes the set of values which can be restored into D_i after w is restored into D_j. Procedure INSERT inserts the values in Δ back into the corresponding active domain. Again to delete an $(n+m)$-variable constraint, we have to delete all $(n+m)$ edges associated with it.

Let $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ be a generalized constraint.
Let (i, C) be an edge associating node i with the constraint C.
Let $AC(G, (i, C), v) = \exists v_{i_1} \in D_{i_1}, \ldots, v_{i_n} \in D_{i_n}, w_{j_1} \in D_{j_1}, \ldots, w_{j_m} \in D_{j_m}$
such that $v_i = v$ and $C([v_{i_1}, \ldots, v_{i_n}], [w_{j_1}, \ldots, w_{j_m}])$.

procedure ARCDEL(**in** (i, C), **out** Δ)
Post: $\Delta = \{v \notin D_i \mid \forall (i, C') \in edge(G), AC(G, (i, C), v)\}$.

procedure LOCALARCDEL(**in** $(i, C), j, w, \Delta$)
Pre: $w \in D_j$.
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_1 = \{v \notin D_i \mid \forall (i, C') \in edge(G), AC(G_{\setminus Q}(i, C), (i, C), v)\}$,
$\Delta_2 = \{v \notin D_i \mid \forall (i, C') \in edge(G), AC(G, (i, C), v)\}$.

Figure 36: Specifications of Parametric Procedures for DELETEEDGE

Correctness

The correctness of the algorithms follows from the correctness of algorithms AC-4 and AC-5, which allows us to conclude that the algorithms are correct for any single execution. However, we also have to show that the data structures which are used to keep the arc consistency information of the network are always updated properly after each execution so that the network can be built incrementally.

The latter property depends on the hierarchical nature of the network. First observe that each edge is considered at most once for each execution of INSERTEDGE or DELETEEDGE because there is no cycle in the network. The information is propagated in one direction only and there is no node which constrains itself through a loop. It is clear that after the execution of ARCINS or LOCALARCINS on an edge, it is arc consistent. Since each edge is only visited at most once, the edge and its supporting data structure will remain intact throughout the rest of the execution of INSERTEDGE.

For DELETEEDGE, we can prove the correctness by a simple induction on the distance of a node from the deleted edge (i, C). The node i is made arc consistent after the initial execution of ARCDEL. Thus, all nodes of distance 0 from (i, C) are arc consistent. Now, assume that all nodes of distance k are arc consistent. If the active domain of a node i' at distance k is altered, all nodes at distance $k + 1$ which are constrained by node i' will be reconsidered by LOCALARCDEL as the edges between the two nodes are put onto the queue. Thus, by induction, all arcs in the network are made arc consistent when the queue is exhausted.

Thus, if we can show that the data structures we use for each class of constraints adhere to their semantics after each execution of the parametric procedures ARCINS, LOCALARCINS, ARCDEL and LOCALARCDEL, we can conclude that the algorithms are correct.

```
procedure ArcIns(in (i, C([i],[j])), out Δ)
  begin
1   Δ := ∅;
2   ∀v ∈ D_i^init do
3     if f_ij(v) ∉ D_j then
4     begin
5       unsupported[i, v] := unsupported[i, v] + 1;
6       if v ∈ D_i then Δ := Δ ∪ {v}
7     end
  end
```

Figure 37: Procedure ArcIns for Functional Constraints

5.4 Functional Constraints

In this section, we consider functional constraints with two variables only, although for uniformity, we use the notation for generalized constraints here also.

Convention 50 Let $C([i],[j])$ be a functional constraint. We use the convention that $f_{ij}(v)$ (resp. $f_{ji}(w)$) denotes the value w (resp. v) such that $C([v],[w])$.

We assume that f takes constant time to evaluate for binary functional constraints.

A data structure *unsupported* can be used for keeping arc consistency information for functional constraints. For each value v in the domain D_i^{init}, we have $unsupported[i,v] =\mid S \mid$, where $S = \{(i, C([i],[j])) \mid f_{ij}(v) \notin D_j\}$. The data structure can be initialized by assigning 0 to every element. The overall space complexity and time complexity for initialization of *unsupported* are both $O(nd)$.

Insertion

The instantiations of procedure ArcIns and LocalArcIns are depicted in Figures 37 and 38 respectively. Procedure ArcIns checks every value v in D_i^{init} and increments the data structure $unsupported[i,v]$ if v is not arc consistent wrt the edge $(i, C([i],[j]))$. We only need to check one value in D_j for each v since the constraints are functional. Also note that the structure *unsupported* is updated even if v is not in the active domain D_i since the value might later be reactivated.

Procedure LocalArcIns increments the count $unsupported[i, f_{ji}(w)]$ and removes the value $f_{ji}(w)$ if it is currently in the active domain D_i, since w can be the only value supporting $f_{ji}(w)$ wrt the edge $(i, C([i],[j]))$.

Deletion

The instantiations of procedure ArcDel and LocalArcDel are depicted in Figures 39 and 40 respectively. Procedure ArcDel checks every value in D_i^{init} and decrements the

```
procedure LocalArcIns(in (i, C([i],[j])), j, w, out Δ)
  begin
1   Δ := ∅;
2   if f_ji(w) ∈ D_i^init then
3   begin
4     unsupported[i, f_ji(w)] := unsupported[i, f_ji(w)] + 1;
5     if f_ji(w) ∈ D_i then Δ := {f_ji(w)}
6   end
  end
```

Figure 38: Procedure LocalArcIns for Functional Constraints

```
procedure ArcDel(in (i, C([i],[j])), out Δ)
  begin
1   Δ := ∅;
2   ∀v ∈ D_i^init do
3     if f_ij(v) ∉ D_j then
4     begin
5       unsupported[i, v] := unsupported[i, v] − 1;
6       if unsupported[i, v] = 0 then Δ := Δ ∪ {v}
7     end
  end
```

Figure 39: Procedure ArcDel for Functional Constraints

counter $unsupported[i, v]$ if a value v outside the active domain D_i was unsupported by the edge $(i, C([i],[j]))$ before the edge was removed from the network. If the count reaches zero, we know that the value v is well supported by all existing edges, and hence we can restore the value into the active domain D_i. Procedure LocalArcDel does the same for the specific value $f_{ji}(w)$ after w has been restored into D_j.

Correctness and Complexity

It is clear that the semantics of the data structure *unsupported* holds after each execution of InsertEdge or DeleteEdge since each time an edge is visited, the parametric procedures update the corresponding counter.

The complexity of algorithm InsertEdge is analysed globally over multiple insertions of binary edges. Each execution of procedure ArcIns is of complexity $O(d)$. Procedure LocalArcIns can be executed at most $O(ed)$ times for multiple insertions, and each execution takes constant time. Thus, the overall complexity for the algorithm InsertEdge is $O(ed)$.

Theorem 51 Algorithm InsertEdge is $O(ed)$ for the insertion of multiple binary functional constraints in a hierarchical network.

```
procedure LocalArcDel(in (i, C([i],[j])), j, w, out Δ)
  begin
1   Δ := ∅;
2   if f_ji(w) ∈ D_i^init then
3   begin
4       unsupported[i, f_ji(w)] := unsupported[i, f_ji(w)] − 1;
5       if unsupported[i, f_ji(w)] = 0 then Δ := {f_ji(w)}
6   end
  end
```

Figure 40: Procedure LocalArcDel for Functional Constraints

A similar argument holds for algorithm DeleteEdge, observing that procedure ArcDel takes $O(d)$ time per execution and procedure LocalArcDel can be executed at most $O(ed)$ times over multiple deletions of edges, where each execution takes constant time.

Theorem 52 Algorithm DeleteEdge is $O(ed)$ for the deletion of multiple binary functional constraints in a hierarchical network.

5.5 Anti-Functional Constraints

We now consider another class of constraints, anti-functional constraints.

Convention 53 Let C be an anti-functional constraint. We denote by $f_{ij}(v)$ (resp. $f_{ji}(w)$) the value w (resp. v) where $\neg C([v],[w])$.

We assume that f can be accessed in constant time for binary constraints.

A data structure for use with anti-functional constraints is shown in Figure 41. The element $unsupported[i,v]$ stores the number of edges $(i, C([i],[j]))$ which do not support the value $v \in D_i^{init}$. The structure $last[i,j]$ stores the value in D_i^{init} that has been removed because it is not supported by any value in D_j. It gives the value ∞ which is not in any domain if every value is supported by D_j. The data structure can be initialized by assigning 0 to every element in *unsupported* and ∞ to every element in *last*. The overall space complexity and time complexity for initialization are $O(nd)$ for *unsupported* and $O(d^2)$ for *last*.

Insertion

The instantiations of procedures ArcIns and LocalArcIns are depicted in Figures 42 and 43 respectively. Procedure ArcIns removes a value from D_i only when D_j consists of a single value, since if there are two or more values, every value in D_i is consistent with at least one value in D_j. Note again that the data structures are updated even if the value v

Syntax

$unsupported[i, v]$: integer.
$last[i, j]$: value.

Semantics

$unsupported[i, v] =\mid S \mid$ where $S = \{(i, C([i], [j])) \mid f_{ij}(v) \notin D_j\}$.
$last[i, j] = v$ where $D_j = \{f_{ij}(v)\}$.

Figure 41: A Data Structure for Anti-Functional Constraints

```
procedure ArcIns(in (i, C([i],[j])), out Δ)
   begin
1     v := f_ji(Min(D_j));
2     if Size(D_j) = 1 and v ∈ D_i^init then
3        begin
4           Δ := {v} ∩ D_i;
5           unsupported[i, v] := unsupported[i, v] + 1
6           last[i, j] := v
7        end
8     else Δ := ∅
   end
```

Figure 42: Procedure ArcIns for Anti-Functional Constraints

has already been removed from a previous iteration. In procedure LocalArcIns, since each edge $C([i], [j])$ can remove at most one value from domain i, we do not have to reconsider an edge if there is already a value removed from D_i which is not consistent wrt this edge (line 1). Otherwise, LocalArcIns calls ArcIns to remove the arc inconsistent value, if any (line 2).

```
procedure LocalArcIns(in (i, C([i],[j])),j,w, out Δ)
   begin
1     if last[i, j] ∈ D_i^init then Δ := ∅
2     else ArcIns((i, C([i],[j])), Δ)
   end
```

Figure 43: Procedure LocalArcIns for Anti-Functional Constraints

```
procedure ArcDel(in (i, C([i],[j])), out Δ)
   begin
1     Δ := ∅;
2     if last[i,j] ∈ D_i^init then
3        begin
4           unsupported[i, last[i,j]] := unsupported[i, last[i,j]] − 1;
5           if unsupported[i, last[i,j]] = 0 then Δ := {last[i,j]};
6           last[i,j] := ∞
7        end
   end
```

Figure 44: Procedure ArcDel for Anti-Functional Constraints

```
procedure LocalArcDel(in (i, C([i],[j])),j,w, out Δ)
   begin
1     ArcDel((i, C([i],[j])), Δ)
   end
```

Figure 45: Procedure LocalArcDel for Anti-Functional Constraints

Deletion

The instantiations of procedures ArcDel and LocalArcDel are depicted in Figures 44 and 45 respectively. Procedure ArcDel checks $last[i,j]$ to see whether there is a value in domain i which is not arc consistent with D_j (line 1), and if so, decrements the counter *unsupported*. The value $last[i,j]$ can be restored into D_i if the counter reaches zero, indicating that it is well supported by all edges. Again, procedure LocalArcDel makes use of procedure ArcDel to restore additional values.

Correctness and Complexity

The data structures *unsupported* and *last* adhere to their semantics after each execution of InsertEdge or DeleteEdge, since we take care to update the fields even when a value is not in the active domain.

The complexity of procedure ArcIns and therefore also procedure LocalArcIns is $O(1)$. For multiple insertions of edges, LocalArcIns can be executed at most $O(ed)$ times, hence the overall complexity of InsertEdge is $O(ed)$.

Theorem 54 Algorithm InsertEdge is $O(ed)$ for the insertion of multiple binary anti-functional constraints in a hierarchical network.

Similary, each execution of procedures ArcDel and LocalArcDel takes constant time. Procedure LocalArcDel can be executed at most $O(ed)$ times for multiple deletions of edges. Thus, the overall complexity of DeleteEdge is $O(ed)$.

Theorem 55 Algorithm DELETEEDGE is $O(ed)$ for the deletion of multiple binary anti-functional constraints in a hierarchical network.

5.6 Monotonic Constraints

Now let us consider monotonic constraints. We first generalize two-variable monotonic constraints to incorporate multi-variables.

Generalization

Definition 56 A constraint $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ is an $(n+m)$-variable *generalized monotonic constraint* between $i_1, \ldots, i_n, j_1, \ldots, j_m$ iff there exists a total ordering on the domain D such that, for any $v_{i_1}, \ldots, v_{i_n}, w_{j_1}, \ldots, w_{j_m} \in D$,

$$C([v_{i_1}, \ldots, v_{i_n}], [w_{j_1}, \ldots, w_{j_m}]) \Rightarrow C([v'_{i_1}, \ldots, v'_{i_n}], [w'_{j_1}, \ldots, w'_{j_m}]),$$

for all $v'_{i_1}, \ldots, v'_{i_n}, w'_{j_1}, \ldots, w'_{j_m} \in D$, where $v'_{i_k} \leq v_{i_k}$ for $1 \leq k \leq n$, and $w'_{j_{k'}} \geq w_{j_{k'}}$ for $1 \leq k' \leq m$.

An example of a generalized monotonic constraint is $C([i,j],[k,l]) : i + j < k + l$.

Representation

The relation conventions and primitive operations required for the incremental algorithms of monotonic constraints are shown in Figure 46. Function f returns the maximum or minimum value of the node k as constrained by the arc (k, C). The result might be a value outside the active domain D_k if another edge is more constraining on k than (k, C). Function CONS gives the set of edges constraining k with the same trend as (k, C), including (k, C) itself. Function f is assumed to take time linear to the number of variables in the constraint, and is constant for binary constraints[4]. Function CONS requires $O(s)$ time, where s is the size of the set of edges returned by the function.

A data structure for use with the monotonic constraints is given in Figure 47. The space complexity of the whole data structure is $O(n+e)$. M_i stores the minimum and maximum values of the currently active domain D_i, as well as the number of edges supporting such values. In $support[i, C]$ we keep the maximum or minimum value of D_i^{init}, depending on the trend of the edge, that is currently supported by edge (i, C). The data structures can be initialized for each node i and each edge (i, C) as follows. $M_i.minValue = \min(D_i^{init})$, $M_i.maxValue = \max(D_i^{init})$, $M_i.minCount = 0$, $M_i.maxCount = 0$, and $support[i, C] =$ UNDEF. Here UNDEF denotes a special value which is not in any domain. The initialization can be done in $O(n+e)$ time for the whole network.

[4]In fact, we can assume that f takes constant time for any n-ary constraints, where n is fixed.

Let $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ be a monotonic constraint.
Let (k, C) be the edge associating node k and constraint C, $k \in \{i_1, \ldots, i_n, j_1, \ldots, j_m\}$.

Relation $\succ_{k,C}$

if $trend(k, C)$ is $\begin{cases} \text{descending} \\ \text{ascending} \end{cases}$, $\succ_{k,C} = \begin{cases} > \\ < \end{cases}$.

Relation $\succeq_{k,C}$

if $trend(k, C)$ is $\begin{cases} \text{descending} \\ \text{ascending} \end{cases}$, $\succeq_{k,C} = \begin{cases} \geq \\ \leq \end{cases}$.

Relation $extremum_{k,C}$

if $trend(k, C)$ is $\begin{cases} \text{descending} \\ \text{ascending} \end{cases}$, $extremum_{k,C} = \begin{cases} \max \\ \min \end{cases}$.

function f**(in** k, C**)**: integer
Post: $f = extremum_{k,C}\{v_k \mid C([v_{i_1}, \ldots, v_{i_n}], [v_{j_1}, \ldots, v_{j_m}])\}$,
where $v_{i_l} \in D_{i_l}$ for $1 \leq l \leq n$, and $v_{j_{l'}} \in D_{j_{l'}}$ for $1 \leq l' \leq m$.

function CONS**(in** k, C**)**: set of edges
Post: CONS $= \{(k, C') \mid trend(k, C') = trend(k, C)\}$.

Figure 46: The Generalized Monotonic Module

Let $C([i_1, \ldots, i_n], [j_1, \ldots, j_m])$ be a monotonic constraint.
Let (i, C) be an edge associating node i and constraint C.

Syntax
$M_i.minValue$: value.
$M_i.maxValue$: value.
$M_i.minCount$: value.
$M_i.maxCount$: value.
$support[i, C]$: value.

Semantics
$M_i.minValue = \min(D_i)$.
$M_i.maxValue = \max(D_i)$.
$M_i.minCount = |S|$ where $S = \{C \mid extremum_{i,C} = \min$ and $support[i, C] = M_i.minValue\}$.
$M_i.maxCount = |S|$ where $S = \{C \mid extremum_{i,C} = \max$ and $support[i, C] = M_i.maxValue\}$.
$support[i, C] = extremum_{i,C}(S)$, where $S = \{v_i \mid C([v_{i_1}, \ldots, v_{i_n}], [v_{j_1}, \ldots, v_{j_m}])\}$,
$v_{i_l} \in D_{i_l}$ for $1 \leq l \leq n$, and $v_{j_{l'}} \in D_{j_{l'}}$ for $1 \leq l' \leq m$.

Figure 47: A Data Structure for Monotonic Constraints

```
procedure ArcIns(in (i, C), out Δ)
  begin
1    Δ := ∅;
2    oldsupport := support[i, C];
3    support[i, C] := f(i, C);
4    if M_i.extremeValue = support[i, C] then
5        if support[i, C] ≠ oldsupport then
6            M_i.extremeCount := M_i.extremeCount + 1
7    else if M_i.extremeValue ≻_{i,C} support[i, C] then
8    begin
9        ∀v s.t. M_i.extremeValue ⪰_{i,C} v ≻_{i,C} support[i, C] do
10           if v ∈ D_i then Δ := Δ ∪ {v};
11       M_i.extremeValue := support[i, C];
12       M_i.extremeCount := 1
13   end
  end
```

Figure 48: Procedure ArcIns for Generalized Monotonic Constraints

Convention 57 Let (i, C) be an edge associating node i and constraint C. We adopt the convention that *extremeValue* stands for *maxValue* (resp. *minValue*) in the fields of array M_i if $trend(i, C)$ is descending (resp. ascending). Similarly, we use *extremeCount* to denote the field *maxCount* (resp. *minCount*) if $trend(i, C)$ is descending (resp. ascending).

Insertion

The instantiations of procedure ArcIns and LocalArcIns for generalized monotonic constraints are depicted in Figures 48 and 49 respectively. Procedure ArcIns updates the array M_i and $support[i, C]$ to reflect how the domain i is constrained by the edge (i, C). Lines 2–3 get the current maximum or minimum value of node i as constrained by (i, C) and saves it in $support[i, C]$. This value is compared to $M_i.extremeValue$ on lines 4 and 7. The count is incremented if the two values are equal (line 6), or it is reset to one and $M_i.extremeValue$ is set to the new value if it is more restricting (lines 11–12). In the latter case, the change is equivalent to narrowing the range of active values in the domain, and all values that become outside the active domain are put into Δ (lines 9–10). Note that in the case where the constraining value is looser than $M_i.extremeValue$, ArcIns returns an empty Δ, and we do not have to alter any information in the network beyond the addition of the edge itself.

Deletion

Specifications of procedure ArcDel and procedure LocalArcDel are shown in Figures 50 and 51 respectively. Procedure ArcDel checks whether domain i can be expanded due to a relaxation on the edge (i, C). The new value of i as constrained by (i, C) is compared

```
procedure LocalArcIns(in (i, C), j, w, out Δ)
   begin
1     ArcIns(i, C, Δ)
   end
```

Figure 49: Procedure LocalArcIns for Generalized Monotonic Constraints

to $M_i.extremeValue$ on line 4, and the count is decremented if they are equal. A zero count indicates that there are no more edges that constrain the active domain of i at this value. Thus, we can recompute the new $M_i.extremeValue$ and $M_i.extremeCount$ information (lines 9–20). This corresponds to relaxing the currently active domain, and the values that are to be restored into the domain are put into Δ on lines 21–22. Note again that we do not have to make any changes to the data structure if the value constrained by (i, C) is looser than $M_i.extremeValue$.

Correctness and Complexity

The correctness of the algorithms follows from the fact that each time any one of the parametric procedures is applied on an edge, we update the data structures M and *support* of the edge correspondingly.

The complexity of procedure InsertEdge is analysed globally over multiple insertions of edges. Each execution of procedure ArcIns takes $O(v)$ time (line 3), where v is the maximum number of nodes connected by an edge. Each edge can be considered $O(vd)$ times in procedure LocalArcIns over multiple insertions. Each execution of LocalArcIns also takes $O(v)$ time, since it just makes use of procedure ArcIns. Thus, the complexity of procedure InsertEdge is $O(v^2ed)$.

For binary constraints, the maximum number of variables allowable in a constraint is two, therefore the complexity becomes $O(ed)$, and we have an optimal algorithm, since we have to look at each edge for each domain value at least once. Also, since v is bounded upwards by n, we have a complexity of $O(n^2ed)$ for constraints with arbitrary number of variables.

Theorem 58 Algorithm InsertEdge is $O(n^2ed)$ for the insertion of multiple generalized monotonic constraints and $O(ed)$ for binary monotonic constraints in a hierarchical network.

A similar argument holds for procedure DeleteEdge. Note also that each iteration of the loop at lines 10–20 of ArcDel takes $O(v)$ time and it can be executed at most e_id times for each node, where e_i is the degree of node i, since the active domain becomes larger with each execution. Thus, the loop is executed at most $\sum_{i \in node(G)} e_id$ times, which takes $O(ed)$ time for binary constraints.

Theorem 59 Algorithm DeleteEdge is $O(n^2ed)$ for the deletion of multiple generalized monotonic constraints and $O(ed)$ for binary monotonic constraints in a hierarchical network.

```
procedure ArcDel(in (i, C), out Δ)
   begin
1     Δ := ∅;
2     oldsupport := support[i, C];
3     support[i, C] := f(i, C);
4     if M_i.extremeValue = oldsupport and oldsupport ≠ support[i, C] then
5     begin
6        M_i.extremeCount := M_i.extremeCount − 1;
7        if M_i.extremeCount = 0 then
8        begin
9           M_i.extremeValue := extremum_{i,C}(D_i^{init});
10          ∀(i, C') ∈ Cons(i, C) do
11          begin
12             support[i, C'] := f(i, C');
13             if M_i.extremeValue ≻_{i,C} support[i, C'] then
14             begin
15                M_i.extremeValue := support[i, C'];
16                M_i.extremeCount := 1
17             end
18             else if support[i, C'] = M_i.extremeValue then
19                M_i.extremeCount := M_i.extremeCount + 1
20          end
21          ∀v s.t. M_i.extremeValue ⪰_{i,C} v ≻_{i,C} oldsupport do
22             if v ∈ D_i then Δ := Δ ∪ {v}
23       end
24    end
   end
```

Figure 50: Procedure ArcDel for Generalized Monotonic Constraints

```
procedure LocalArcDel(in (i, C), j, w, out Δ)
   begin
1     ArcDel(i, C, Δ)
   end
```

Figure 51: Procedure LocalArcDel for Generalized Monotonic Constraints

6 Conclusion

The generic arc consistency algorithm AC-5 can be instantiated to give $O(ed)$ algorithms for anti-functional constraints and the piecewise counterparts to functional, anti-functional and monotonic constraints. The same approach to AC-5 is applied to formulate a generic path consistency algorithm PC-5 which can be instantiated to PC-4 as well as algorithms with lower complexity for functional and monotonic constraints. Generic incremental arc consistency algorithms are constructed in the framework of generalized multi-variable constraints in a hierarchical network. Instantiations are given for binary functional, anti-functional constraints, and also generalized monotonic constraints. The incremental algorithms have a complexity of $O(ed)$ for constraints with fixed number of variables, which is optimal for these constraint subclasses.

Acknowledgements

Special thanks to Yves Deville for reviewing part of the paper. Special special thanks to my advisor Pascal Van Hentenryck for his patience and invaluable advice, without which my work would only be chaos.

References

[1] Y. Deville and P. Van Hentenryck. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In *International Joint Conference on Artificial Intelligence*, Sidney, Australia, August 1991.

[2] C.C. Han and C.H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[3] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[4] A.K. Mackworth and E.C. Freuder. The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[5] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[6] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[7] R.E. Tarjan. Amortized Computational Complexity. *SIAM Journal of Algebraic Discrete Methods*, 6:306–318, 1985.

[8] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[9] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc consistency algorithm and its specializations. To Appear in *Artificial Intelligence.*

[10] D. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI271, MIT, MA, November 1972.