Constraint Satisfaction over Connected Row Convex Constraints ⋆

Yves Deville, Olivier Barette and Pascal Van Hentenryck

Université catholique de Louvain,
Pl. Ste Barbe 2, B-1348 Louvain-la-Neuve, Belgium
{yde,ob,pvh}@info.ucl.ac.be

Abstract

This paper studies constraint satisfaction over connected row convex (CRC) constraints. It shows that CRC constraints are closed under composition, intersection, and transposition, the basic operations of path-consistency algorithms. This establishes that path consistency over CRC constraints produces a minimal and decomposable network and is thus a polynomial-time decision procedure for CRC networks. This paper also presents a new path-consistency algorithm for CRC constraints running in time $O(n^3d^2)$ and space $O(n^2d)$, where n is the number of variables and d is the size of the largest domain, improving the traditional time and space complexity by orders of magnitude. The paper also shows how to construct CRC constraints by conjunction and disjunction of a set of basic CRC constraints, highlighting how CRC constraints generalize monotone constraints and presenting interesting subclasses of CRC constraints. Experimental results show that the algorithm behaves well in practice.

1 Introduction

Constraint satisfaction techniques have been found useful in many areas such as combinatorial optimization, hardware design, robotics, knowledge bases, and temporal reasoning to name only a few. Some applications require to find one or all solutions, in which case consistency techniques (e.g., arc and path consistency) are instrumental in reducing the size of the search space. Other applications require to put the constraints network in minimal form, e.g., to remove redundant information, in which case consistency techniques apply as well since they remove values which cannot appear in solutions.

In recent years, increasing attention has been devoted to the study of special classes of constraints or constraint networks. These studies are motivated both

⋆ This paper is an extended version of [DBVH97]

Preprint submitted to Elsevier Science *26 February 1999*

by practical considerations (e.g., constraint languages are based on a set of primitive constraints) and by theoretical considerations, since stronger results and more efficient algorithms can be obtained by exploiting special properties and tractable classes of constraints can be identified.

The research described in this paper was motivated by the class of row-convex constraints identified by van Beek and Dechter [vBR95]. When the constraints of a path-consistent constraint network are row-convex (or can be made row-convex by permutation of values in the domain), then the constraint network is minimal and decomposable and a solution can be found without backtracking in $O(n^2d)$ after application of a path-consistency algorithm (which runs in $O(n^3d^3)$). Unfortunately, row-convex constraints are not closed under composition and intersection, the main operations of path-consistency algorithms. As a consequence, no conclusion can be drawn a priori for a constraint network of row-convex constraints, since its path-consistent subnetwork may or may not be row-convex.

The first contribution of this paper is the definition of a a new class of constraints, called connected row-convex (CRC) constraints, which is closed under the operations of path-consistency algorithms. As a consequence, the class of CRC constraints is shown to be tractable. The paper also shows how to construct CRC constraints by conjunction and disjunction of a set of basic CRC constraints, highlighting how CRC constraints generalize monotone constraints [Mon74] and presenting interesting subclasses of CRC constraints.

The second contribution of the paper is a path-consistency algorithm, called PC-CRC, tailored to CRC constraints and running in $O(n^3d^2)$ time and in $O(n^2d)$ space. PC-CRC improves traditional algorithms by an order of magnitude and is a decision procedure for networks of CRC constraints. The algorithm is obtained by instantiating a generic path-consistency algorithm PC-GEN. Such an approach facilitates the understanding of the algorithm, provides a framework for the description and comparison of existing path-consistency algorithms, and can be reused for the development of new (specialized or not) path-consistency-like algorithms.

The rest of the paper is organized as follows. Section 2 introduces the necessary background and Section 3 discusses related work. Section 4 describes the class of CRC constraints and shows that this class is tractable. Section 5 presents the generic algorithm PC-GEN which is then instantiated to CRC constraints in Section 6. Section 7 provides analysis and experimental results. Section 8 concludes the paper. Additional detail on some of the presented results can be found in [Bar97].

2 Preliminaries

Definition 1 (Binary constraint network [Mon74])

A (binary) constraint network $\mathcal{N} = (Var, D, C)$ is a set Var of n variables $\{1, \ldots, n\}$ represented by natutal numbers, a finite domain D_i of possible values for each variable i (the set D is the union of all domains), and a set C of binary constraints bewteen variables. A constraint between variable i and j, denoted by C_{ij}, is a set of couples ($C_{ij} \subseteq D_i \times D_j$) that specifies the allowed pairs of values for i and j.

The fact that $(v, w) \in C_{ij}$ is also denoted by $C_{ij}(v, w)$. Given a constraint network $\mathcal{N} = (Var, D, C)$, d will denote the size of the largest domain, and $arc(\mathcal{N})$ the set $\{(i, j) \mid C_{ij} \in C\}$. We assume the existence of a total ordering over D. It is finally required that $(v, w) \in C_{ij}$ iff $(w, v) \in C_{ji}$. As usual, a constraint C_{ij} will also be seen as a Boolean matrix with $\mid D_i \mid$ rows and $\mid D_j \mid$ columns. The Boolean value are represented by 0 and 1 for convenience. Rows and columns are ordered according to the underlying order over D. A 1 (resp. 0) at position (v, w) in the matrix means $(v, w) \in C_{ij}$ (resp. $(v, w) \notin C_{ij}$). To simplify the presentation, each domain D_i is also represented by a (pseudo-binary) constraint C_{ii} such that $C_{ii}(v, v)$ holds iff $v \in D_i$. Domain D_i and constraint C_{ii} can be used in an interchangeable way.

Consistency techniques aim at reducing the size of the problem without altering its set of solutions. Such techniques are usually called *local* consistency as they analyze different *parts* of the problem and remove elements that cannot belong in a solution of the problem.

Definition 2 *$\langle v_1, \ldots, v_n \rangle$ is a solution of $\mathcal{N}$ iff $C_{ij}(v_i, v_j)$ holds for all $(i, j) \in arc(\mathcal{N})$.*

Definition 3 *Two constraint networks $\mathcal{N}$ and $\mathcal{N}'$ are* equivalent *iff $\mathcal{N}$ and $\mathcal{N}'$ have the same solutions.*

The following definition describe path consistency of constraint networks [Mac77]) .

Definition 4 *A constraint network $\mathcal{N} = (Var, D, C)$ is path-consistent iff, for every triple (i, k, j) of variables, we have that for every $v_i \in D_i$ and $v_j \in D_j$ such that $C_{ij}(v_i, v_j)$, there exists $v_k \in D_k$ such that $C_{ik}(v_i, v_k)$ and $C_{kj}(v_k, v_j)$.*

Note that if the definition of path consistency does allow identical nodes (i, k, i), then path consistency implies arc consistency. The purpose of a path-consistency algorithm is to compute, given a constraint network $\mathcal{N} = (Var, D, C)$, an equivalent constraint network $\mathcal{N}' = (Var, D', C')$ which is path-consistent. The resulting constraint network will thus also be arc-consistent.

We can draw a parallel between path- and arc-consistency algorithms. An arc-consistency algorithm removes arc inconsistent values from the domains of variables. Hence the outputs of an arc-consistency algorithm are domains. Working on domains is not sufficient for a path-consistency algorithm. Suppose that $D_i = D_j = \{a, b\}$. It can be the case that $\langle a, b\rangle$ is path inconsistent for some path (i, k, j). Such a path inconsistency does not mean that a (or b) should be removed from D_i (or D_j) but that, in a solution, it is impossible to have $\langle a, b\rangle$ as value for the couple of variables i, j. Hence a path-consistency algorithm should "remove" path-inconsistent tuples from constraints, and the output should be constraints. Such algorithms usually handle explicit representation of constraints and assume a complete constraint network. An incomplete constraint network can be easily transformed into a complete one by adding *TRUE* constraints (constraints allowing any combination of values) between every pair of variables (i, j) where $(i, j) \notin arc(\mathcal{N})$.

Definition 5 *A constraint network $\mathcal{N}$ is* minimal *iff*
$\forall i, j \in arc(\mathcal{N})\ \forall v, w \in D\ :\ $ if $C_{ij}(v, w)$ then there is a solution of $\mathcal{N}$ with values v and w assigned to i and j.

Definition 6 *A constraint network $\mathcal{N}$ is* decomposable *iff, $\forall v_{i_1} \dots v_{i_k}$ satisfying all the constraints relating nodes $i_1 \dots i_k (1 \leq k < n)$ and for any new node i_{k+1}, there exists $v_{i_{k+1}}$ such that $v_{i_1} \dots v_{i_k}, v_{i_{k+1}}$ satisfy all the constraints relating nodes $i_1 \dots i_k, i_{k+1}$.*

A decomposable constraint network is also called strongly n-consistent [Fre82]. Decomposable constraint networks have thus the property that any consistent instantiation of some variables can be extended to a solution, without backtracking. A decomposable constraint network is of course minimal. In a minimal constraint network, it is not possible to prune further the constraints without removing solutions.

3 Related Work

This research was motivated by van Beek's result on row-convex constraint. A constraint C_{ij} is *row-convex* if, in each row of its matrix representation, all the ones are consecutive. Van Beek and Dechter [vBR95] show that, when the constraints of a path-consistent constraint network are row-convex (or can be made row-convex by permutation of values in the domain), then the constraint network is minimal and decomposable. One can thus compute a solution without backtracking in $O(n^2 d)$. Solving the CSP can then be done in $O(n^3 d^3)$, the time complexity of the PC algorithm. Unfortunately, row-convex constraints are not closed under composition and intersection. As a consequence, no conclusion can be drawn a priori for a constraint network of row-convex constraints, since its path-consistent subnetwork may or may not be row-convex.

This paper proposes a subclass which is closed under the main operations of path-consistency algorithms. Different subclasses are already presented in [vBR95]. It covers binary relations on domains with two elements (graph 2-coloring), and linear binary constraints which is a particular cases of CRC constraints. Closed classes are also analysed and identified in [JC95,JCC98], where Jeavons and Cooper identify the class of max-closed constraints that can be solved in polynomial time ($O(n^4d^4)$ for binary constraints). Our class of CRC constraints, which can be solved in $O(n^3d^2)$, intersects with max-closed constraints, but is not a subset. The authors alos presents implicational relations and also other tractables constraints not based on row convexity. Montanari [Mon74] already shows that a path-consistent tree or distributive networks are minimal. He also shows that path consistency of (total) monotone constraints produces a decomposable network. Note that CRC constraints are not distributive and generalize the monotone functions of Montanari.

The class of CRC constraints is also related to discrete temporal reasoning [vB92b]. Valdès-Peres [VP87] shows that path-consistency algorithms find the minimal network for a subclass of Allen's interval algebra [All83]. Such a result has also been proposed in the context of point algebra [VK86,vB89].

The idea of row convexity has also been exploited in the context of continuous constraints [HF94,HF96]. They start from the result that, when constraints are convex and binary, path-consistency is sufficient to ensure decomposability. They show that for continuous domains, this result can be generalized to ternary and n-ary constraints using some other notion of consistency ((3,2)-relational consistency).

4 Connected Row-Convex Constraints

This section introduces CRC constraints, a particular case of row-convex constraints. CRC constraints are preserved by path-consistency algorithms (i.e., the application of a path-consistency algorithms on a CRC network produces a CRC-network), which is not the case of general row convex constraints. As a consequence, applying path consistency on CRC-constraints produces a minimal and decomposable network. In this section, we use the matrix representation of constraints. Given the initial domains D_i and D_j, a constraint C_{ij} can be represented by a Boolean matrix. We assume a total ordering of the elements in the domains. The rows and columns are ordered according to the underlying order of the domain.

4.1 Row-Convex Constraints

Van Beek introduced the concept of row-convex constraint [vB92a].

Definition 7 *A constraint C_{ij} is* row-convex *if, in each row of the matrix representation of C_{ij}, all the ones are consecutive (i.e., no two ones within a single row are separated by a zero in that same row.*

In [vB92a], van Beek showed that if the constraints of a path-consistent constraint network are row-convex (or can be made row-convex by permutation of values in the domain), then the constraint network is minimal and decomposable. One can thus compute a solution without backtracking.

The problem is that the class of row-convex constraints is too large as row convexity can be lost during the path-consistency algorithm. Van Beek suggested to restrict the class of row-convex constraints to a class closed under composition, intersection, and transposition, the basic operations in PC algorithms. Following this suggestion, we present in the next section such a class of row-convex constraints.

4.2 CRC Constraints

Row-convex constraints exhibits two problems during path-consistency algorithms. First, when a row-convex constraint is composed of disjoint blocks of 1s, its composition with another row-convex constraint may not be row-convex. Second, even if disjoint blocks are forbidden, intersection may create empty rows and columns and thus disjoint blocks. Here is an illustration of these two problems :

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cap \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

CRC constraints avoid both problems. Informally, a constraint is CRC if, after removing the empty rows, it is row-convex and connected (two successive rows either intersect or are consecutive).

Definition 8 *The* reduced form *of a constraint C_{ij}, denoted by C^*_{ij}, is obtained by removing all the empty rows and columns in its matrix representation. The* domain *of i through the constraint C_{ij}, denoted by $D_i(C_{ij})$, is the set $\{v \in D \mid \exists w : \langle v, w\rangle \in C_{ij}\}$*

Definition 9 *Let C_{ij} be a row-convex constraint and $v \in D_i(C_{ij})$. The* image *of v in C_{ij} is the set $\{w \mid \langle v, w\rangle \in C_{ij}\}$. Because of the row convexity of C_{ij}, this set is represented as an interval $[w_1, w_m]$ (over the domain $D_j(C_{ji})$) and we denote w_1 and w_m by $min(C_{ij}, v)$ and $max(C_{ij}, v)$ respectively. We also denote by $succ(w, D_j(C_{ji}))$ and $pred(w, D_j(C_{ji}))$ the successor and the predecessor of w in $D_j(C_{ji})$. For ease of notation, these two operations will be denoted $succ(w)$ and $pred(w)$ when there is no ambiguity on the underlying domain.*

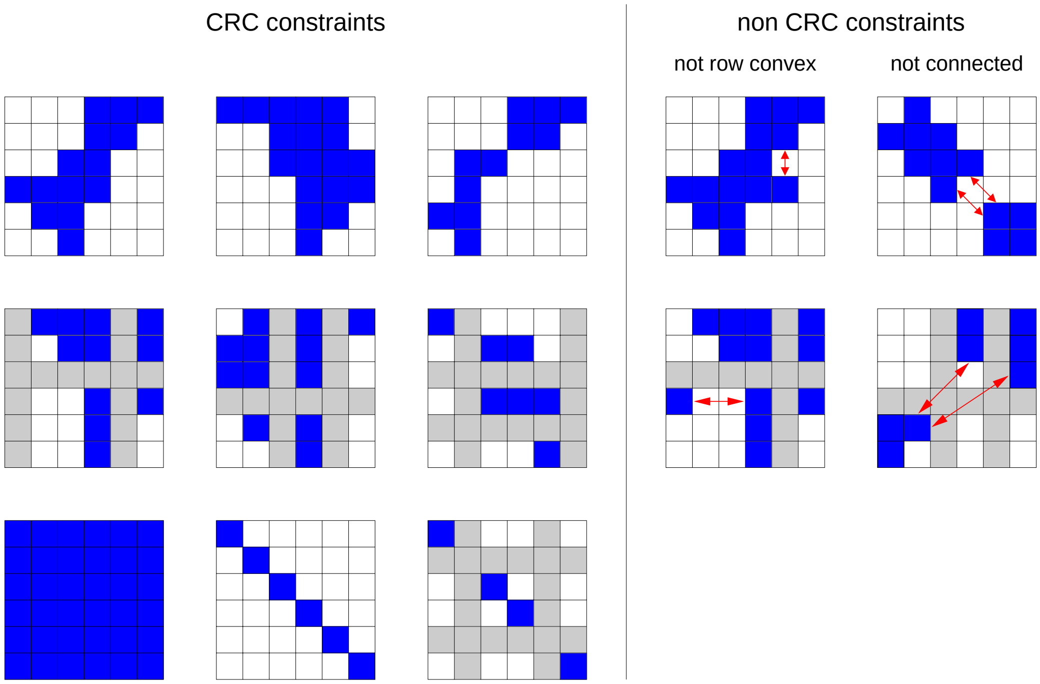

Fig. 1. Examples CRC constraints

Definition 10 *A row-convex constraint* C_{ij} *is* connected *iff the images* $[a, b]$ *and* $[a', b']$ *of two consecutive rows in* C^*_{ij} *is such that*

$$b' \geq pred(a) \wedge a' \leq succ(b)$$

Definition 11 *A constraint* C_{ij} *is* connected row-convex *(CRC) iff*

(i) C^*_{ij} *and* C^*_{ji} *are row-convex,*
(ii) C^*_{ij} *and* C^*_{ji} *are connected.*

We assume that C_{ij} is always the transposition of C_{ji}. Examples of CRC constraints are given in Figure 4.2 (1 are in black, empty rows/columns are in grey). Notice that CRC constraints are not necessarily row-convex (because of empty rows) and that row-convex constraints are not necessarily CRC (not connected rows). The top right constraint in Figure 4.2 is an example showing that a CRC constraint cannot always be made CRC by permutations of rows and columns.

It is interesting to notice that, in the definition of CRC, the second condition can be simplified, as suggested by the following property.

Theorem 12 *Assuming that* C^*_{ij} *and* C^*_{ji} *are row-convex,* C^*_{ij} *is connected iff* C^*_{ji} *is connected.*

Proof. Let C^*_{ij} and C^*_{ji} be row-convex. Suppose C^*_{ij} not connected. A simple case analysis on the cause of the non connectivity of C^*_{ij} leads to the non connectivity of C^*_{ji}. □

4.3 Properties of CRC Constraints

This section shows that CRC constraints are closed under composition, intersection and transposition.

Lemma 13 *The deletion of rows and columns in a CRC constraint produces a CRC constraint.*

Proof. It is sufficient to prove that the suppression of one (non empty) row to C_{ij} preserve the CRC property. Let v the corresponding element, and C'_{ij} be the resulting matrix. We observe that C'^*_{ij} has exactly one row less, and possibly less columns than C^*_{ij}. It is easy to see that C'^*_{ij} and C'^*_{ji} are row-convex.
Removing a row does not affect the fact that C'^*_{ji} is connected. The images in C_{ji} which contained v has now one less element in C'_{ji}. If the interval becomes empty, the corresponding row is simply suppressed.
Let $[a_1, b_1], [a_2, b_2]$ be the images in C'_{ji} of the rows preceding and following the suppressed row. If these interval were not connected (say because $b_2 < pred(a_1)$), then the columns of C^*_{ij} corresponding to position $succ(b_2), \ldots, pred(a_1)$ are empty, except at row v. Otherwise C^*_{ij} would not be row-convex. Hence removing row v in C_{ij} induces that these columns will be suppressed in C'^*_{ij}. The intervals $[a_1, b_1], [a_2, b_2]$ are thus connected in C'^*_{ij}. □

Lemma 14 *Let C_{ij} be a CRC constraint. Let v_1, v, v_2 be in $D_i(C_{ij})$ such that $v_1 < v < v_2$ and their respective images are $[a_1, b_1]$, $[a, b]$ and $[a_2, b_2]$ in C_{ij}.*

$$b_2 < a_1 \Rightarrow [a, b] \cap [b_2, a_1] \neq \emptyset$$

$$a_2 > b_1 \Rightarrow [a, b] \cap [b_1, a_2] \neq \emptyset$$

$$b_2 \geq a_1 \wedge a_2 \leq b_1 \Rightarrow [a_1, b_1] \cap [a_2, b_1] \subseteq [a, b]$$

Theorem 15 *The intersection of two CRC constraints is a CRC constraint.*

Proof. Let A_{ij} and B_{ij} be two CRC constraints. Let $C_{ij} = A_{ij} \cap B_{ij}$.
If A_{ij} or B_{ij} have empty rows or columns, we may suppress in A_{ij} *and* in B_{ij} all rows and columns which are empty either in A_{ij} or in B_{ij}, and repeat this process until no more rows or columns can be suppressed. The elements in C_{ij} not in the intersection of the obtained reduced matrices are obviously null. We may thus assume that A_{ij} and B_{ij} have no empty rows or columns.

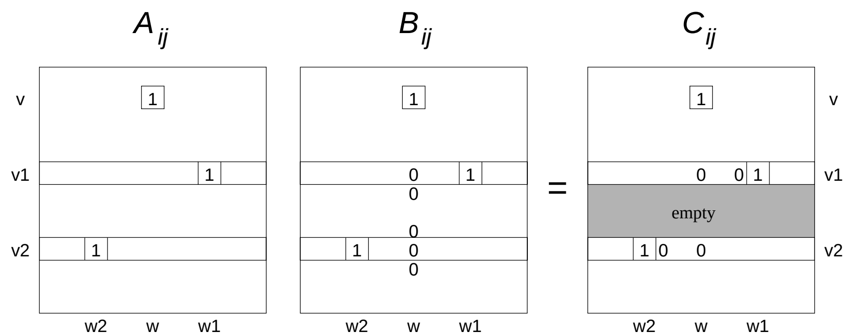

Fig. 2. Intersection of two CRC constraints

Suppress in A_{ij} and B_{ij} all rows and columns which are empty either in A_{ij} or in B_{ij}. Repeating this process until no more rows or columns can be suppressed. Let A_{ij} and B_{ij} be the resulting matrices, and C_{ij} be $A_{ij} \cap B_{ij}$. The elements of C_{ij} not in C_{ij} are obviously null. It is thus sufficient to show that C_{ij} is CRC.

By Lemma 13, A_{ij} and B_{ij} are CRC. The row convexity of C_{ij} (and C_{ji}) is obvious as each row (and column) is the intersection of intervals.

Let $v_1, v_2 \in D_i(C_{ij})$ such that v_1 and v_2 have non empty rows in C_{ij}, the rows between v_1 and v_2 are empty, and row v_1 and row v_2 are not connected, as illustrated in Figure 4.3. Let the leftmost 1 in row v_1 be at position w_1, and the rightmost 1 in row v_2 be at position w_2. The other possible cases are symmetrical. We show that all the columns between w_2 and w_1 are empty. Hence that C_{ij} is CRC.

Assume that such a column is not empty (e.g., $C_{ij}(v, w) = 1$).

We necessarily have a 1 at positions (v_1, w_1), (v_2, w_2) and (v, w) in A_{ij} and in B_{ij}. As $C_{ij}(v_1, w) = 0$, either A_{ij} or B_{ij} has a 0 at position (v_1, w). Without loss of generality, we suppose that $B_{ij}(v_1, w) = 0$. By row convexity of B_{ij}, all elements below (v_1, w) are also null in B_{ij}. The matrix B_{ij} is then *not connected* somewhere between row v_1 and row v_2. This is impossible as B_{ij} is CRC. $\square$

Theorem 16 *The composition of two CRC constraints is a CRC constraint.*

Proof. Let $C_{ij} = C_{ik}.C_{kj}$.

Empty rows in C_{ik} and empty columns in C_{kj} can be removed as producing empty rows/columns in C_{ij}. An empty column in C_{ik} can be suppressed together with its corresponding row in C_{kj} without affecting the result. Similarly for empty rows in C_{kj}. Repeating this process leads to two constraints included in C^*_{ik} and C^*_{kj}. By Lemma 13, these two constraints are CRC constraints. We may thus assume that C_{ik} has no empty rows, and C_{kj} no empty columns.

Let us first show that $C_{ij} = C_{ik}.C_{kj}$ is row-convex. Let $v_1 < v < v_2$ such that $C_{ij}(v_1, w) = 1$ and $C_{ij}(v_2, w) = 1$. Let $[a_1, a'_1]$, $[a, a']$ and $[a_2, a'_2]$ be the images of v_1, v, and v_2 in C_{ik}. Let $[b, b']$ be the image of w in C_{jk}. We have

$$[a_1, a'_1] \cap [b, b'] \neq \emptyset$$
$$[a_2, a'_2] \cap [b, b'] \neq \emptyset$$

From the application of Lemma 14 on a simple case analysis on the relative positions of $[a_1, a'_1]$ and $[a_2, a'_2]$, we can conclude that $[a, a'] \cap [b, b'] \neq \emptyset$, hence that $C_{ij}(v, w) = 1$.

Let us now prove that C_{ij} is CRC. Let $v_1, v_2 \in D_i$ such that v_1 and v_2 have non empty rows in C_{ij}, the rows between v_1 and v_2 are empty, and rows v_1 and v_2 are not connected, as illustrated in Figure 4.3. Let the leftmost 1 in row v_1 be at position w_1, and the rightmost 1 in row v_2 be at position w_2. The other possible cases are symmetrical. We show that all the columns between w_2 and w_1 are empty. Hence that C_{ij} is CRC.

Assume that such a column is not empty (e.g., $C_{ij}(v, w) = 1$ and $C_{ij}(v, succ(w)) = 0$).

From $C_{ij}(v_1, w_1)$, there exists some u_1 such that $C_{ik}(v_1, u_1) = 1$, $C_{kj}(u_1, w_1) = 1$. As $\langle v_1, w_1 \rangle$ is the leftmost 1, $C_{kj}(u_1, b) = 0$ for $b < w_1$. By the row convexity of C_{ij}, $\langle v_1, w_1 \rangle$ is also the lowest 1. Hence $C_{ik}(a, u_1) = 0$ for $a > v_1$.

From $C_{ij}(v_2, w_2)$, there exists some u_2 such that $C_{ik}(v_2, u_2) = 1$, $C_{kj}(u_2, w_2) = 1$. As $\langle v_2, w_2 \rangle$ is the rightmost 1, $C_{kj}(u_2, b) = 0$ for $b > w_2$. By the row convexity of C_{ij}, $\langle v_2, w_2 \rangle$ is also the highest 1. Hence $C_{ik}(a, u_2) = 0$ for $a > v_2$.

From $C_{ij}(v, w)$, there exists some u such that $C_{ik}(v, u) = 1$, $C_{kj}(u, w) = 1$. As $\langle v, w \rangle$ is the downmost 1, $C_{ik}(a, u) = 0$ for $a > v$. Given that C_{ik} is CRC, we must have $u < u_1 < u_2$. By the row convexity of C_{kj}, $C_{kj}(c, w) = 0$ for $c \geq u_1$. This makes C_{kj} *not* connected somewhere between rows u_1 and u_2. Impossible as C_{kj} is CRC.

The proof for the symmetrical cases is similar. □

Theorem 17 *The transposition of a CRC constraint is a CRC constraint.*

Theorem 18 *Let $\mathcal{N}$ be composed of CRC constraints. The application of a path-consistency algorithm to $\mathcal{N}$ produces a minimal and decomposable constraint network.*

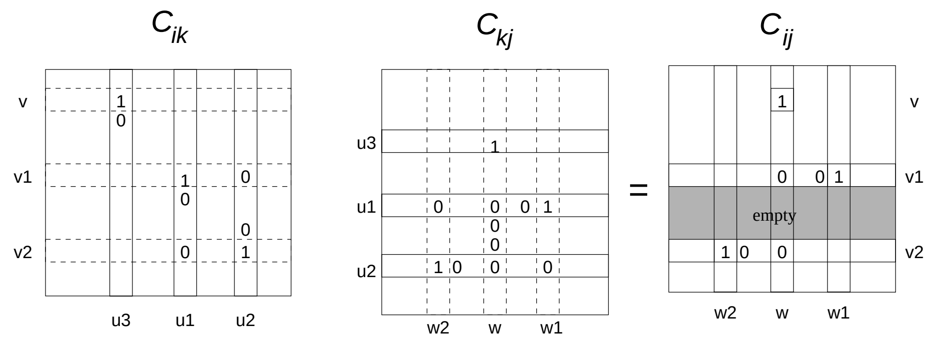

Fig. 3. Composition of two CRC constraints

Proof. Straightforward as path-consistency can be achieved by only using composition, intersection and transposition of (the matrix representation of) constraints. □

Theorem 19 *The class of CRC constraints is tractable.*

4.4 Examples of CRC Constraints

It is important to discuss some examples of CRC constraints and to show how they generalize monotone constraints [Mon74]. Let us assume the existence of a (total) ordering in each domain D_i. For ease of notation, we will use the same ordering symbol $\leq$ for all the domains.

Definition 20 *Let* $\preceq$ *and* $\succeq$ *be total orderings on* D_i *and* D_j*, respectively. A (binary) constraint* C_{ij} *is* $(\preceq, \succeq)$*-monotone if*

- $\forall v, v' \in D_i,\ \forall w \in D_j\ :\ if\ C_{ij}(v,w)\ and\ v' \preceq v\ \ then\ C_{ij}(v',w)$
- $\forall v \in D_i,\ \forall w, w' \in D_j\ :\ if\ C_{ij}(v,w)\ and\ w' \succeq w\ \ then\ C_{ij}(v,w')$

A constraint is monotone if it is $(\leq, \geq)$-monotone. It is possible to generalize the class of monotone constraints by allowing any combination of the ordering relations. This provides some insights on why CRC constraints are important and how they generalize monotone constraints.

Definition 21 *A constraints is* staircase *if it is* (α, β)*-monotone with* $\alpha, \beta \in \{\leq, \geq\}$.

Examples of staircase constraints are : $ax + by + c \leq 0$, $ax + by + c \geq 0$, $axy + b \leq 0$, $axy + b \geq 0$, $af(x) + by + c \leq 0$, $af(x) + by + c \geq 0$, with a, b, c rationals, $f(x)$ a function such that $f'(x)$ is either always positive or always negative on the considered interval. Intersection and/or composition of staircase constraints are CRC but not necessarily staircase. For instance, assuming a domain $D = \{1..10\}$, the two constraints

$$5x - 3y - 4 \geq 0 \;\; \wedge \;\; 2x - y - 7 \leq 0$$
$$x.y \leq 10 \;\; \wedge \;\; x + y \geq 0$$

are CRC but not staircase. It is also possible to define other (sub)classes of CRC constraints, such as $y \geq (ax + by + c)^2$, with b integer, and assuming a domain of positive integers. These constraints are CRC, but not staircase.

Staircase constraints are an important generalization of monotone constraints and are tractable.

Proposition 22 *The class of staircase constraints is tractable.*

The difference between monotone constraints and CRC constraints appears clearly if a contructive definition of CRC constraints is given. This definition involves conjunctions and disjunctions of *basic* CRC constraints. Intuitively, a basic constraint defines a rectangle within the domain, or it defines an empty row/column.

Definition 23 *A* basic CRC constraint *between variables i and j is a constraint of one of the following forms :*

$$\begin{array}{lrcl}
\text{(Upper Right)} & UR_{ij}^{ab}(v,w) & = & v \leq a \;\wedge\; w \geq b \\
\text{(Upper left)} & UL_{ij}^{ab}(v,w) & = & v \leq a \;\wedge\; w \leq b \\
\text{(Lower Right)} & LR_{ij}^{ab}(v,w) & = & v \geq a \;\wedge\; w \geq b \\
\text{(Lower Left)} & LL_{ij}^{ab}(v,w) & = & v \geq a \;\wedge\; w \leq b
\end{array}$$

A basic domain constraint *is a constraint of the form*

$$\begin{array}{lrcl}
\text{(Domain)} & DC_i^a(v) & = & v \neq a
\end{array}$$

Notice that a $(\leq, \geq)$-monotone constraint over a domain D can also be expressed as a disjunction of Upper Right basic constraints. The next definition, and its associated theorem, thus show clearly the generalization provided by CRC constraints. The definition provides a constructive definition of CRC constraints.

Definition 24 *A* CNF-CRC constraint *is a constraint of the form :*

$$(\bigvee_{\substack{a_k \in D_i \\ b_k \in D_j}} UR_{ij}^{a_k b_k}) \wedge (\bigvee_{\substack{a_k \in D_i \\ b_k \in D_j}} UL_{ij}^{a_k b_k}) \wedge (\bigvee_{\substack{a_k \in D_i \\ b_k \in D_j}} LR_{ij}^{a_k b_k}) \wedge (\bigvee_{\substack{a_k \in D_i \\ b_k \in D_j}} LL_{ij}^{a_k b_k})$$
$$\wedge \;\; (\bigvee_{a_k \in D_i} DC_i^{a_k}) \wedge (\bigvee_{b_k \in D_j} DC_j^{b_k})$$

Theorem 25 *The following classes of constraints are tractable and equivalent :*

(i) CRC constraints,
(ii) CNF-CRC constraints,

(iii) *the closure, by intersection and composition, of staircase constraints and basic domain constraints.*

5 PC-GEN: a Generic Path-Consistency Algorithm

In this section we present a new generic path-consistency algorithm PC-GEN that can be parametrized like the arc consistency algorithm AC-5 [VDT92]. This approach has many advantages. The generic algorithm can be instantiated to existing path-consistency algorithms, providing thus a framework for the description and comparison of existing algorithms. New path-consistency algorithms can also be derived from the generic one. Only the two procedures PathCons and LocalPathCons have to be implemented. The correctness of the obtained instantiation is then a consequence of the correctness of the generic algorithm. This approach is used in the next section to design PC-CRC, an efficient path-consistency algorithm specialized to CRC constraints.

5.1 Basic Operations

The specification of the basic operations in PC-GEN are given in Figure 4. All specifications assume a constraint network $\mathcal{N} = (Var, D, C)$. A parameter p subscripted with 0 (p_0) represents the value of p at call time. As is traditional, PC-GEN uses a queue Q to drive the algorithm. A tuple $\langle i, k, j, v\rangle$ in Q implies that it is necessary to reconsider the constraint C_{ij} wrt path (i, k, j) knowing that, for some u, $\langle v, u\rangle$ has been removed from C_{ik}. Procedure Enqueue is required to take $O(s)$ time, where s is the number of new elements to insert in the queue and procedure Dequeue must take constant time. The deletion of tuples is performed by procedure Prune, which removes tuple $\langle v, w\rangle$ from C_{ij} and $\langle w, v\rangle$ from C_{ji}. Hence,

$$\langle v, w\rangle \in C_{ij} \iff \langle w, v\rangle \in C_{ji}$$

will be an invariant of the algorithm, assuming it holds initially.

5.2 Parametric Procedures

PC-GEN is parametrized by two procedures (Figure 5), PathCons and LocalPathCons whose implementations are left open. Procedure PathCons computes the set Δ of tuples in C_{ij} which are not path-consistent for the path (i, k, j). Because of the relationship between C_{ij} and C_{ji}, Δ is also the set of tuples (in reverse order) of C_{ji} that are not path-consistent for path (j, k, i). This is illustrated in Figure 5.2(a)

Procedure LocalPathCons returns in Δ a set of tuples of C_{ij} that are not path-consistent for (i, k, j) after tuple $\langle v, u\rangle$ (for some u) has been removed from the constraint C_{ik}. The set Δ is also the set of tuples (in reverse order)

procedure PRUNE(**in** Δ, i, j)
Pre: $i, j \in arc(\mathcal{N})$.
Post: $C_{ij} = C_{ij_0} \setminus \{\langle v, w\rangle \mid \langle v, w\rangle \in \Delta\}$,
$C_{ji} = C_{ji_0} \setminus \{\langle w, v\rangle \mid \langle v, w\rangle \in \Delta\}$.

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

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

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

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

Fig. 4. The basic operations for PC-GEN

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

procedure PATHCONS(**in** i, k, j, **out** Δ)
Pre: $i, k, j \in arc(\mathcal{N})$.
Post: $\Delta = \Delta_2$, with
$\Delta_2 = \{\langle v, w\rangle \in C_{ij} \mid \neg PC_{ikj}(v, w)\}$

procedure LOCALPATHCONS(**in** i, k, j, v, **out** Δ)
Pre: $i, k, j \in arc(\mathcal{N})$.
Post: $\Delta_1 \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_1 = \{\langle v, w'\rangle \in C_{ij} \mid \neg PC_{ikj}(v, w')\}$.
$\Delta_2 = \{\langle v', w'\rangle \in C_{ij} \mid \neg PC_{ikj}(v', w')\}$.

Fig. 5. Parametric procedures for PC-GEN

of C_{ji} that are not path-consistent in path (j, k, i) after tuple $\langle u, v\rangle$ has been removed from C_{ki}.

The size of Δ computed by LOCALPATHCONS can vary. The set Δ_1, illustrated in Figure 5.2(b), contains the tuples in C_{ij} that become path inconsistent for (i, k, j) due to the removal of tuple $\langle v, u\rangle$ from C_{ik}. In some cases, it is possible, but not always desirable, to prune a larger set of tuples. As an extreme case, Δ_2 prunes all tuples in C_{ij} which are path inconsistent wrt (i, k, j) at call time, regardless of whether they can be supported by $\langle v, u\rangle$ (see Figure

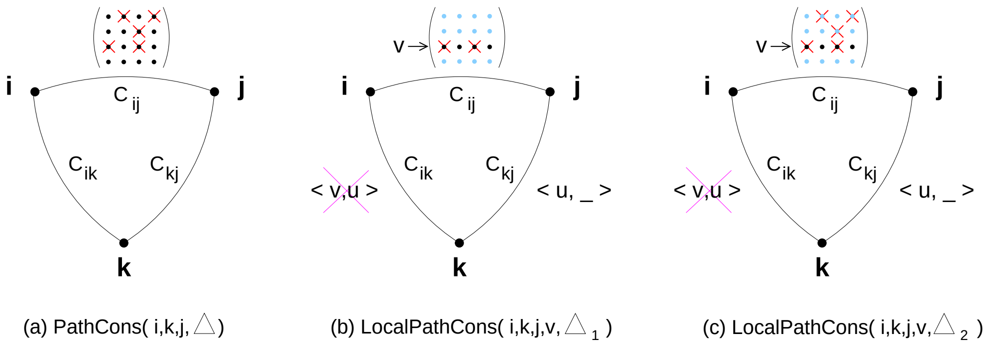

Fig. 6. Pruning of PathCons and LocalPathCons

5.2(c)). The specification of the parametric procedures takes advantage of this fact and allows for both flexibility and efficiency. Any intermediate Δ can be computed.

Notice that the definition of $PC_{ikj}(v, u)$ (Figure 5) does not require $u \in D_k$. This comes from the simple observation that the fixpoint of

$$C_{ij} := C_{ij} \cap C_{ik}.C_{kk}.C_{kj}$$

is the same as the fixpoint of

$$C_{ij} := C_{ij} \cap C_{ik}.C_{kj}$$

computed for all $i, j, k \in arc(\mathcal{N})$.

The choice of not considering C_{kk} will simplify the instantiation of these procedures for particular classes of constraints, without affecting the correctness of PC-GEN.

5.3 *Algorithm* PC-GEN

PC-GEN is depicted in Figure 7 and mimics AC-5. In the loop on lines 2–7, procedure PathCons identifies the path-inconsistent tuples with respect to each path of length two. The inconsistent tuples are enqueued 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}. The restriction $i \leq j$ in the first loop is justified by the fact that PathCons(i, k, j, Δ) treats both paths (i, k, j) and (j, k, i). Note that paths of the form (i, i, i) could be discarded since the resulting Δ set is empty. The removal of the tuple $\langle v, w\rangle$ in C_{ij} and $\langle w, v\rangle$ in C_{ji} requires to reconsider all length-two paths involving either (i, j) or (j, i) as the first or as the second

```
Algorithm PC-GEN
   Post: N is a path-consistent constraint network equivalent to N_0.
   begin
1     InitQueue(Q);
2     for each i,k,j ∈ arc(N) with i ≤ j do
3     begin
4        PathCons(i,k,j,Δ);
5        Enqueue(i,j,Δ,Q);
6        Prune(Δ,i,j)
7     end;
8     while not EmptyQueue(Q) do
9     begin
10       Dequeue(Q,i,k,j,v);
11       LocalPathCons(i,k,j,v,Δ);
12       Enqueue(i,j,Δ,Q);
13       Prune(Δ,i,j)
14    end
   end
```

Fig. 7. The Path-Consistency Algorithm PC-GEN

arc. It is however unnecessary to consider explicitly the second arc (in the Enqueue procedure) since LocalPathCons(i, j, k, v) covers both paths (i, j, k) and (k, j, i) and LocalPathCons(j, i, k, w) covers paths (j, i, k) and (k, i, j). This is because of the invariant maintained by procedure Prune.

5.4 Correctness

The correctness of PC-GEN is given in Appendix A.1.

Theorem 26 *Algorithm PC-GEN terminates and is correct.*

5.5 Complexity Bounds

Although we do not develop here a concrete implementation for the basic operations of PC-GEN, we may assume the complexity bound of $O(1)$ for Dequeue, $O(\Delta)$ for prune, and $O(s)$ for Enqueue, where s is the number of elements to insert in the queue. As usual the O notation denotes an upper bound of the worst case complexity.

If the complexity of PathCons is $O(t)$, the loop at lines 2–7 takes $O(n^3) \cdot O(t)$ time. If PathCons takes $O(\Delta)$ time, the loop at lines 2–7 has a complexity of $O(q)$, where q is the total number of elements that can be enqueued throughout the execution of PC-GEN. Also, if LocalPathCons takes $O(t)$ time (with $O(t) \geq O(d)$), the loop at lines 8–14 takes $O(q) \cdot O(t)$ time. Finally, if LocalPathCons takes $O(\Delta)$ time, the loop at lines 8–14 has a complexity of $O(q)$, These observations will become helpful when we will analyze particular instances of PC-GEN.

Theorem 27 *Given a time complexity of $O(d^2)$ for procedure* PathCons *and a time complexity of $O(d)$ for procedure* LocalPathCons*, algorithm* PC-GEN *is bounded by $O(n^3 d^3)$.*

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-GEN *is bounded by $O(n^3 d^2)$.*

5.6 Relaxing the Specification of the Parametric Procedures

The specification of the generic procedures PathCons and LocalPathCons can be further relaxed without affecting the correctness nor the complexity of PC-GEN. Such a generalisation is important as it formalizes existing path-consistency algorithms such as PC-4, and also allows an efficient specialisation of PC-GEN for CRC conctraints. The general idea is that, when some $\langle v, w\rangle$ is not path-consistent wrt (i, k, j) (i.e., $\neg PC_{ikj}(v, w)$), it is not necessary to prune $\langle v, w\rangle$ immediately if we are ensured that $\langle v, w\rangle$ will eventually be pruned when some other element in the queue will be processed.

Definition 29 *The tuple $\langle v, w\rangle$ is* look-ahead-1 *(LH(1)) for path (i, k, j) iff*

$$\langle i, k, j, v\rangle \in Q \vee \langle j, k, i, w\rangle \in Q$$

Definition 30 *The tuple $\langle v, w\rangle$ is* look-ahead-m *(LH(m)) for path (i, k, j) $(m > 1)$ iff*

$$\begin{aligned}&\exists u : \langle v, u\rangle \in C_{ik} \;\wedge\; \exists k' : \neg PC_{ik'k}(v, u) \;\wedge\; (\langle v, u\rangle \textit{ is LH(m-1) for } ik'k)\\ \vee\; &\exists u : \langle u, w\rangle \in C_{jk} \;\wedge\; \exists k' : \neg PC_{jk'k}(u, w) \;\wedge\; (\langle u, w\rangle \textit{ is LH(m-1) for } jk'k)\end{aligned}$$

The relaxed parametric procedures are specified in Figure 8. We will denote by PC-GEN* the algorithm PC-GEN using the procedures PathCons* and LocalPathCons*. The correctness of PC-GEN* is proven in the Appendix A.2.

One may also extend the queue by considering tuples of the form $\langle i, k, j, \langle v, w\rangle\rangle$. Such a tuple denotes it is necessary to reconsider constraint C_{ij} wrt to path (i, k, j) because $\langle v, w\rangle$ has been removed from constraint C_{ik}. Such an extension is useful for instantiating PC-GEN to PC-4.

The specification of procedures Dequeue and Enqueue can easily be extended. A tuple $\langle v, w\rangle$ will now be LH(1) for path (i, k, j) iff

$$\exists u : \langle i, k, j, \langle v, u\rangle\rangle \in Q \vee \langle j, k, i, \langle w, u\rangle\rangle \in Q$$

Let $PC_{ikj}(v,w) = \exists u : \langle v,u\rangle \in C_{ik}$ and $\langle u,w\rangle \in C_{kj}$.
$PC^*_{ikj}(v,w) = PC_{ikj}(v,w) \ \vee\ \exists m : \langle v,w\rangle$ is LH(m) for ikj

procedure PathCons*(**in** i, k, j, **out** Δ)
Pre: $i,k,j \in arc(\mathcal{N})$.
Post: $\Delta_2^* \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_2^* = \{\langle v,w\rangle \in C_{ij} \mid \neg PC^*_{ikj}(v,w)\}$
$\Delta_2 = \{\langle v,w\rangle \in C_{ij} \mid \neg PC_{ikj}(v,w)\}$

procedure LocalPathCons*(**in** i,k,j,v, **out** Δ)
Pre: $i,k,j \in arc(\mathcal{N})$.
Post: $\Delta_1^* \subseteq \Delta \subseteq \Delta_2$, with
$\Delta_1^* = \{\langle v,w'\rangle \in C_{ij} \mid \neg PC^*_{ikj}(v,w')\}$.
$\Delta_2 = \{\langle v',w'\rangle \in C_{ij} \mid \neg PC_{ikj}(v',w')\}$.

Fig. 8. Relaxed parametric procedures for PC-GEN

With the given specification of LocalPathCons, such an extension of the queue is useless as only the element v is used in the definition of the resulting Δ set [1].

5.7 *Instantiating PC-GEN to Existing PC Algorithms*

One can show that PC-GEN can be instantiated to yield a PC algorithm with a time complexity of $O(n^3d^3)$, and a space complexity of $O(n^3d^2)$. Such complexities were obtained in [Sin95,Chm96]. The classical PC-4 has the same time complexity, but a space complexity of $O(n^3d^3)$.

PC-GEN can also be instantiated to existing path-consistency algorithms, providing thus a framework for their comparison. For instance, PC-GEN can be instantiated to PC-2 [Mac77] and PC-6 [Chm96]. The classical PC-4 [HL88] is an instance of PC-GEN* using the extended queue. It is here necessary to use PC-GEN* instead of PC-GEN, as PC-4 uses a technique covered by our definition of LH(1).

6 PC-CRC: a Path-Consistency Algorithm for CRC Constraints

In this section, we provide PC-CRC, an efficient instance of PC-GEN specialized to CRC constraints. PC-CRC has a time complexity of $O(n^3d^2)$ and a space complexity of $O(n^2d)$. We describe the representation of CRC constaints and the instanciation of the generic procedures. A precise and complete de-

[1] The value u could be used as follows in the specification of LocalPathCons (resp. LocalPathCons*). The set Δ_1 (resp. Δ_1^*) can be further reduced by imposing $\langle u,w'\rangle \in C_{kj}^{init}$ (where C_{kj}^{init} denotes the original set of constraint tuples between i and j).

Let $D = \{b, \ldots, B\}$.
Let $C_{ij} = \{\langle v_1, v_1\rangle, \ldots, \langle v_m, v_m\rangle\}$ if $i = j$
$= \{\langle v_1, w_1\rangle, \ldots, \langle v_m, w_m\rangle\}$ if $i \neq j$ (where $v_k, w_k \in D$)

Data Structure

Syntax

$C_{ij}.supmin$: array $[b..B]$ of element $\in D$.
$C_{ij}.supmax$: array $[b..B]$ of element $\in D$.
$C_{ij}.first$: element $\in D$.
$C_{ij}.succ$: array $[b..B]$ of element $\in D$.
$C_{ij}.pred$: array $[b..B]$ of element $\in D$.

Semantics

$C_{ij}.supmin[v] = min(C_{ij}, v)$
$C_{ij}.supmax[v] = max(C_{ij}, v)$
$C_{ij}.first = min\{v \in D_i(C_{ij})\}$
$C_{ij}.succ[v] = succ(v)$ in $D_i(C_{ij})$
$C_{ij}.pred[v] = pred(v)$ in $D_i(C_{ij})$

Invariant

$C_{ij} = C_{ji}^T$
$C_{ij}.supmin[v] \in D_j(C_{ji})$
$C_{ij}.supmax[v] \in D_j(C_{ji})$

Interface

Let $PC_{ikj}^2(v, w) = PC_{ikj}(v, w) \;\; \vee \;\; \exists m \leq 2 \; : \; \langle v, w\rangle$ is LH(m) for ikj

function EmptySupport(**in** v,w, i,k,j): Boolean
Post: EmptySupport(v,w, i,k,j) = $\neg PC_{ikj}^2(v, w)$

function First(**in** i,j): Integer
Post: First(i,j) = $min\{v \in D_i(C_{ij})\}$

function Min(**in** v, i,j): Integer
Post: Min(v, i,j) = $min(C_{ij}, v)$

function Max(**in** v, i,j): Integer
Post: Max(v, i,j) = $max(C_{ij}, v)$

function Succ(**in** v, i,j): Integer
Post: Succ(v, i,j) = $succ(v)$ in $D_i(C_{ij})$

function Pred(**in** v, i,j): Integer
Post: Pred(v, i,j) = $pred(v)$ in $D_i(C_{ij})$

Fig. 9. The CRC Constraint Module

scription will be provided. As the application of PC-CRC produces a minimal and decomposable constraint network, we also provide an algorithm to find a solution of the constraint network.

6.1 Representation of CRC Constraints

CRC constraints can be represented in space $O(d)$ as shown in Figure 9. It is necessary to keep a description of $D_i(C_{ij})$, since row convexity is only

```
procedure PathCons(in i, k, j, out Δ)
   begin
      Δ := ∅;
      for each v ∈ D_i(C_ij) do
      begin
         LocalPathCons(i, k, j, v, Δ_v);
         Δ := Δ ∪ Δ_v;
      end
   end

procedure LocalPathCons(in i, k, j, v, out Δ)
   begin
      BoundedMin(i, k, j, ⟨v, Max(v, i,j) ⟩, Δ', w_min);
      if w_min =Max(v, i,j) then Δ := Δ'
      else
      begin
         BoundedMax(i, k, j, ⟨v, Min(v, i,j) ⟩, Δ'', w_max);
         Propagate(i,j,k,⟨v, w_min⟩, BoundedMin, pred, Δ_1);
         Propagate(i,j,k,⟨v, w_min⟩, BoundedMin, succ, Δ_2);
         Propagate(i,j,k,⟨v, w_max⟩, BoundedMax, pred, Δ_3);
         Propagate(i,j,k,⟨v, w_max⟩, BoundedMax, succ, Δ_4);
         Δ := Δ' ∪ Δ'' ∪ Δ_1 ∪ Δ_2 ∪ Δ_3 ∪ Δ_4;
      end
   end
```

Fig. 10. PathCons and LocalPathCons for CRC constraints.

enforced on the reduced form. Figure 9 also specifies the operations on CRC constraints which are all implemented in constant time. For instance, EmptySupport(v, w, i, k, j) can be implemented by $b' \geq a \wedge a' \leq b$ with a =Min(v, i, k), b =Max(v, i, k), a' =Min(w, j, k), and b' =Max(w, j, k). As the domains $D_k(C_{ki})$ and $D_k(C_{kj})$ are not necessarily identical, the EmptySupport(v, w, i, k, j) does not compute $PC_{ikj}(v, w)$, but $PC^2_{ikj}(v, w)$, which is $PC^*_{ikj}(v, w)$ with LH(m) restricted to $m \leq 2$.

6.2 *Instantiation of the Generic Procedures*

An implementation of Procedures PathCons and LocalPathCons is given in Figure 10. Note that PathCons is expressed in terms of LocalPathCons. In LocalPathCons, BoundedMin computes the interval Δ' to be removed on the left of the interval in row v while BoundedMax computes the interval Δ'' to be removed on the right of the interval in row v. Although this pruning is sufficient, it may destroy the CRC property. We know that removing *all* the inconsistent tuples yields a CRC constraint. To preserve the property, we thus perform additional pruning on the rows above or below v. This is the role of the Propagate instructions. The specifications and implementations of the subproblems procedures are given Appendix A.3. The intuition behind LocalPathCons is captured in Figure 11. Because $C_{ij} := C_{ij} \cap C_{ik}.C_{kj}$ produces a CRC constraint, the implementation is guaranteed to keep C_{ij} connected row-convex. Note that Propagate works from v to the exterior, while BoundedMin and BoundedMax work from the exterior to the interior.

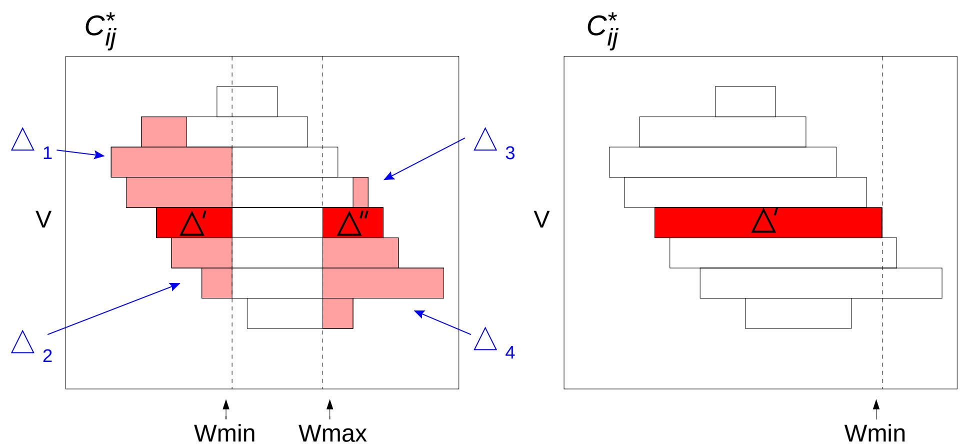

Fig. 11. Illustrating LocalPathCons for CRC constraints : two possible cases.

The implementation of LocalPathCons could be optimized in several ways. For instance, in Figure 11, there is an element *above* v, left to $Wmin$, which is supported. As the resulting constraint is known to be CRC, every element *below* v, left to $Wmin$, can directly be be suppressed.

6.3 Correctness

The LocalPathCons procedure for CRC constraints is an instance of the LocalPathCons* procedure specified in Figure 8, where LH(m) has been restricted to the case $m \leq 2$. Lines 1 and 5 compute the set Δ_1^* which is sufficient for correctness. In order to keep the CRC property, the sets $\Delta_1, \Delta_2, \Delta_3$ and Δ_4 are then computed in lines 6–10. We have $\Delta_i \subseteq \Delta_2^*$. Since $\Delta_2^* \subseteq \Delta_2$, we have $\Delta_i \subseteq \Delta_2$.

The correctness of PathCons is a direct consequence of the correctness of LocalPathCons.

6.4 Complexity

Prune can be performed in $O(\Delta)$ assuming the elements of Δ are ordered to preserve the CRC property, as specified in Figure 12. The ordering can be performed during the construction of Δ during LocalPathCons without incurring any cost. An implementation of Δ as a doubly-linked list is sufficient for this purpose given the way Δ is constructed as mentioned in the

procedure $\text{P}{\scriptsize\text{RUNE}}$(**in** Δ, i, j)
Pre: $i, j \in arc(\mathcal{N})$,
C_{ij} is a CRC constraint,
$C_{ij} \setminus \Delta$ is a CRC constraint.
Post: $C_{ij} = C_{ij_0} \setminus \{\langle v, w\rangle \mid \langle v, w\rangle \in \Delta\}$,
$C_{ji} = C_{ji_0} \setminus \{\langle w, v\rangle \mid \langle v, w\rangle \in \Delta\}$.

Fig. 12. Pruning for PC-CRC.

previous section. The complexity of Procedures PROPAGATE, BOUNDEDMIN and BOUNDEDMAX is obviously $O(\Delta)$. Hence LOCALPATHCONS is $O(\Delta)$. By Theorem 28, the time complexity of PC-GEN is $O(n^3d^2)$. The space complexity per constraint is $O(d)$ and $O(nd)$ for all the constraints. The space complexity of the queue is bounded by $O(n^2d)$ because elements in the queue can be grouped as tuples of the form $\langle i, j, E, v\rangle$, where the set E is initially $arc(\mathcal{N}) \setminus \{j\}$. The set E can be shared by all elements of the queue except the first one.

Theorem 31 *For CRC constraints, PC-GEN has a time complexity of $O(n^3d^2)$ and a space complexity of $O(n^2d)$.*

The above theorem is valid for incomplete constraint networks of CRC constraints as well, since the completion of the constraint network introduces *TRUE* constraints which are CRC.

6.5 Finding a Solution

A path-consistent constraint network with CRC constraints is decomposable due to Helly's theorem (e.g., [HF96]). The proof in [vBR95] is constructive and the author proposes a $O(n^2d)$ algorithm to find a solution. We propose in Figure 13 an INSTANTIATE procedure with a time complexity of $O(n^2)$ for CRC constraints. It is based on van Beek's algorithm, but takes advantage of the data structure.

The total complexity to detect inconsistency or to find a solution of a constraint network composed with CRC constraints is thus $O(n^3d^2)$, the time complexity of the path-consistency algorithm.

Theorem 32 *The class of CRC constraints is tractable in $O(n^3d^2)$.*

7 Analysis and Experimental Results

This section analysis the class of CRC networks. It also studies how PC-CRC performs in practice (does it perform better than the theoretical complexity? How large are the constant factors?). Extensive experimentations have

```
procedure Instantiate(in 𝒩, out ⟨x_1, ..., x_n⟩)
Pre: 𝒩 has only CRC constraints, and is path-consistent,
      D_i ≠ ∅ (1 ≤ i ≤ n)
Post: ⟨x_1, ..., x_n⟩ is a solution of 𝒩.

   begin
      for i := 1 to n do
      begin
         L := First(i,i);
         for j := 1 to i-1 do L := max( L, Min(x_j,j,i) ) ;
         x_i := L
      end
   end
```

Fig. 13. Instantiate for CRC Constraints.

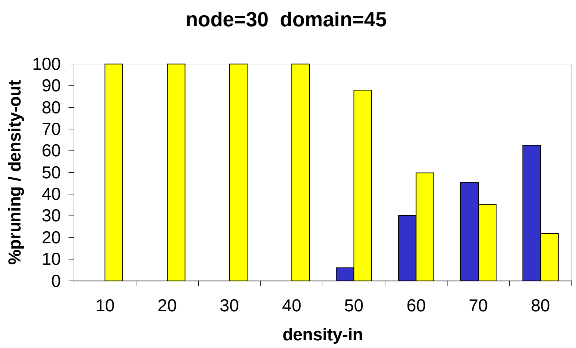

Fig. 14. Pruning of PC-CRC

been performed. Data sets have been randomly generated for the following combinations of the parameters:

- n (number of node) : from 10 to 80;
- d (size of the domain) : from 10 to 45;
- *density* : from 10% to 80%.

Density is here defined as the probability that $C(v, w)$ holds for $v, w \in D$ (i.e the number of ones in the matrices compared to the size of the matrices). Only complete constraint networks were considered and more than 2,000 executions of PC-CRC have been recorded and analyzed using statistical methods. All the experiments have been performed on a SUN Ultra 1 workstation running Solaris.

7.1 Satisfiable vs non-Satisfiable Constraint Networks

We first analyse CRC constraint networks from the satisfiability point of view. As PC-CRC produces a minimal and decomposable constraint network, if the

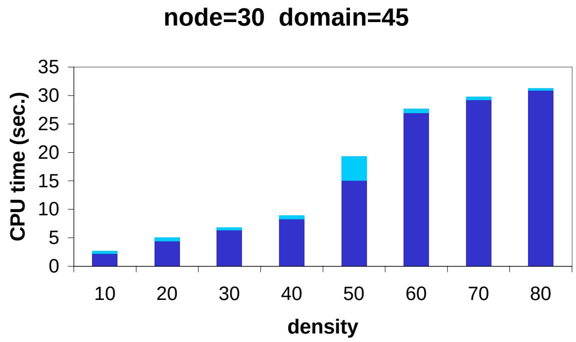

Fig. 15. Execution time for different densities

algorithm terminates without detecting an inconsistency, then the constraint network is known to be satisfiable. Figure 7 depicts the pruning for $n = 30$, $d = 45$, and densities from 10 to 80. The dark bars measure the density of the constraints after application of PC-CRC (*density-out*). The grey bars indicate the pruning factor ((*density-in* − *density-out*) / *density-in*). Non satisfiable networks thus have a pruning factor of 100%. For all the different values of *density-in*, the statistical error of the resulting *density-out* is less that 2.4 (i.e., the 95% confidence interval is included in *density-out* ±2.4).

From these experiments, one can observe that when *density-in* is less than 45, the constraint network is always non-satisfiable. When *density-in* is greater than 55, the constraint network is always satisfiable. Between 45 and 55, the pourcentage of satisfiable constraint networks is around 53%. The global shape of the results also holds for other combinations of n and d, except for the position of the frontier between the non-satisfiable and satisfiable problems. In our data sets, the frontier always lies between 40 and 60.

7.2 Influence of Density on Complexity

The theoretical time complexity of PC-CRC is $O(n^3 d^2)$. This complexity could be refined to take into account the density of the constraint network. We then have a time complexity of $O(n^3\ (density * d)^2)$.

It is interesting to compare this new theoretical complexity with experimental results. Figure 7.2 displays the execution time of PC-CRC for $n = 30$, $d = 45$ and various densities. The top of the dark bar denotes the lower bound of the 95% confidence interval and the top of the grey bar the upper bound. This shows a significative difference of execution time between non-satisfiable (density 10 to 45) and satisfiable (density 55 to 80) constraint networks. Interestingly, the execution time for satifiable constraint networks is almost independent from the density.

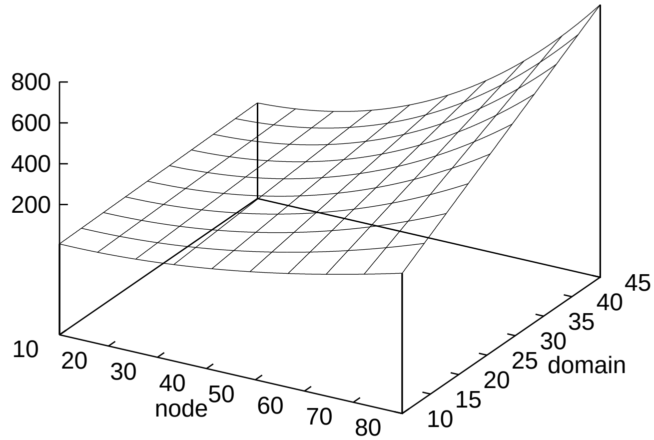

Fig. 16. Execution time for different densities

7.3 Theoretical Complexity vs Experimental Complexity

The theoretical time complexity of $O(n^3d^2)$ only provides an *upper bound* of the *worst-case* complexity. By experimental complexity, we mean to model the real execution time of a set of test problems by a polynomial of the form:

$$\sum a_{ij}n^id^j \quad \text{(with } i,j \geq 0\text{, and } i+j \leq 5\text{)}$$

The degree 5 is infered by the theoretical complexity.

Such an experiment has been performed for a density of 70, since it is representative of the difficult cases. We used a statistical software package called ECHIP. This software proposed an experimental plan (number of constraint networks to generate, values of n and d to consider). For the measured execution times, the software proposed the following complexity :

$$2.23 * 10^{-5}n'^3d' + 0.00333n'^2d' + 0.0772n'^2 + 0.154n'd' + 3.82n' + 2.79d' + 59.29$$

Only the statically significative coefficients a_{ij} are considered, $n' = n - 35$ and $d' = d - 17.5$. The ECHIP software were also able to assess both the validity and the predictive ability of the model.

These experiments show that the time complexity of PC-CRC is n^3d, and that the actual coefficient of the polynomial are very small for the higher degree terms (the first term is dominant only for $n > 185$). The CPU time of the experiments is shown in Figure 7.3. As can be observed, the CPU time is linear wrt d for a given n.

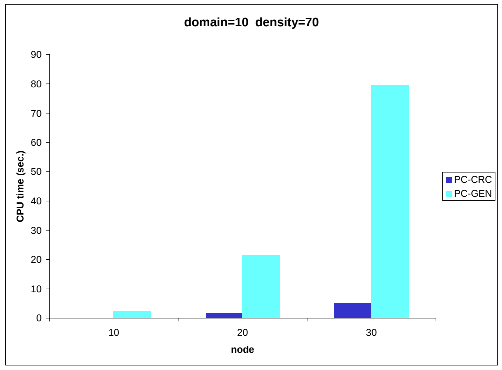

Fig. 17. PC-CRC vs PC-GEN

7.4 PC-CRC vs Classical PC Algorithms

For solving CRC constraint networks, one may use the specialized PC-CRC algorithm or any other PC algorithm. Although we know the theoretical complexity of PC-CRC is better than the theoretical complexity of classical PC algorithms, and that the experimental complexity of PC-CRC is very good, it is intersting to analyse the experimental complexity of general PC algorithms on CRC constraint networks. To perform this experimentation, we used an instance of PC-GEN close to PC-4, but with a better space complexity. We compare this algorithm and PC-CRC for $d = 10$, a density of 70, and $n = 10, 20, 30$ (See Figure 7.3). The confidence intervals of the execution times for both algorithms are very small (always less than 5% of the measured execution time). The results clearly indicates that, in this case, the experimental complexity of the general algorithm is worse than PC-CRC. Similar differences appear for other values of the parameters.

8 Conclusion

This paper introduces the class of CRC constraints and showed that it is closed under composition, intersection, and transposition, the basic operations of path-consistency algorithms. As a consequence, path consistency over CRC constraints produces a minimal and decomposable network and is thus a polynomial-time decision procedure for CRC networks. This paper then presented a new path-consistency algorithm for CRC constraints running in time $O(n^3d^2)$ and space $O(n^2d)$, where n is the number of variables and d is the size of the largest domain, improving the traditional time and space complexity by orders of magnitude. Experimental results show that the algorithm behaves well in practice. The paper also showed how to construct CRC constraints by conjunction and disjunction of a set of basic CRC constraints, highlighting how CRC constraints generalize monotone constraints, presenting interesting subclasses of CRC constraints, and highlighting how to construct CRC constraints. The automatic recognition of CRC constraint constraint networks, i.e.,

> "given a constraint network, does there exist an ordering on the domains that makes the constraint network CRC?"

remains an interesting open issue. To be useful, an algorithm answering this question should run in time $\Omega(n^3d^2)$ since otherwise it is preferable to apply a general path-consistency algorithm (running in time $O(n^3d^3)$) and to apply an algorithm recognizing row-convex constraint constraint networks (which runs in time $O(n^3d^2)$ [vBR95]). Finally, current work is devoted to studying how to use similar ideas for other classes of discrete and continuous constraints and for other consistency notions (e.g. [FE96].

9 Acknowledgments

We thank anonymous IJCAI and AIJ reviewers for helpful comments. Special thanks to Christine Jacqumot for her help in the development and analysis of the experimental plans. This research is partially supported by the *Actions de recherche concertées (ARC/95/00-187)* of the Direction générale de la Recherche Scientifique – Communauté Française de Belgique and by an NSF National Young Investigator Award with matching funds.

References

[All83] J.F. Allen. Maintaining Knowledge About Temporal Reasoning. *Journal of the ACM*, 26:832–843, 1983.

[Bar97] O. Barette. Un algorithme de chemin-consistence et son instanciation à une classe de réseaux décomposable. Mémoire de fin d'études, Département d'Ingénierie Informatique, Université catholique de Louvain (Belgium), June 1997.

[Chm96] A. Chmeiss. Sur la consistance de chemin et ses formes partielles. In *Actes du Congrès AFCET-RFIA'96, Rennes*, 1996.

[DBVH97] Y. Deville, O. Barette, and P Van Hentenryck. Constraint satisfaction over connected row convex constraints. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 405–410, 1997.

[FE96] E. Freuder and C.D. Elfe. Neighborood inverse consistency preprocessing. In *AAAI-96*, Portland, Oregon, 1996.

[Fre82] E.C. Freuder. A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, 29:24–32, 1982.

[HF94] D. Haroud and B. Faltings. Global consistency for continuous constraints. In *Principles and Practice of Constraint Programming*, pages 40–50. Springer-Verlag LNCS 874, 1994.

[HF96] D. Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints, An International Journal*, 1:85–118, 1996.

[HL88] C.C. Han and C.H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[JC95] P.G. Jeavons and M.C. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79:327–339, 1995.

[JCC98] P. Jeavons, D. Cohen, and M. Cooper. Constraints, Consistency and Closure. *Artificial Intelligence*, 101:251–265, 1998.

[Mac77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(2):95–132, 1974.

[Sin95] M. Singh. Path consistency revisited. In *IEEE-ICTAI'95, Washington DC*, 1995.

[vB89] P. van Beek. Approximation algorithms for temporal reasoning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 745–750, 1989.

[vB92a] P. van Beek. On the minimality and decomposability of constraint network. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92)*, pages 447–452, 1992.

[vB92b] P. van Beek. Reasoning About Qualitative Temporal Reasoning. *Artificial Intelligence*, 58:297–326, 1992.

[vBR95] P. van Beek and Dechter R. On the minimality and global consistency of row convex networks. *Journal of the ACM*, 42(3):543–561, May 1995.

[VDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2–3):291–321, October 1992.

[VK86] M. Vilain and Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-86), Philadelphia*, pages 132–144, 1986.

[VP87] R.E. Valdès-Pérez. The satisfiability of temporal constraint network. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87), Seattle*, pages 745–750, 1987.

Appendix

A.1 Correctness of PC-GEN

The correctness of PC-GEN is proved using a similar argument than in [VDT92]. Given two constraint networks $\mathcal{N} = (Var, D, C)$ and $\mathcal{N}' = (Var, D', C')$, we define $\mathcal{N}' \sqsubseteq \mathcal{N}$ if $\forall i, j \in arc(\mathcal{N}) D_i \subseteq D'_i \wedge C_{ij} \subseteq C'_{ij}$. We also define $\mathcal{N}'' = \mathcal{N}' \sqcup \mathcal{N}$, with $\mathcal{N}'' = (Var, D'', C'')$, $D''_i = D_i \cup D'_i$ and $C''_{ij} = C_{ij} \cup C'_{ij}$.

We prove that the output of PC-GEN is the largest path-consistent constraint network for $\mathcal{N}$. One can easily show that such a largest constraint network always exists, is unique, and is equivalent to $\mathcal{N}$. We first show that the invariant $\mathcal{N}^* \sqsubseteq \mathcal{N}$ is preserved in PC-GEN, where $\mathcal{N}^*$ is the largest path-consistent constraint network for $\mathcal{N}$. Partial correctness (i.e. if the program terminates, it produces a correct result) can then be proved by showing that, when PC-GEN terminates, the constraint network is path-consistent. We finally prove termination, hence the (total) correctness of the algorithm.

Lemma 33 *Let $\mathcal{N}^*$ be the largest path-consistent constraint network for $\mathcal{N}_0$. After the execution of PC-GEN, we have $\mathcal{N}^* \sqsubseteq \mathcal{N}$.*
Proof. We prove a stronger result: The invariant $\mathcal{N}^* \sqsubseteq \mathcal{N}$ is preserved in PC-GEN at lines 2 and 8. The invariant holds for the first execution of line 2, as $\mathcal{N} = \mathcal{N}_0$ and $\mathcal{N}^* \sqsubseteq \mathcal{N}_0$. Execution of lines 4 to 6 preserves the invariant because Δ contains path-inconsistent tuples that cannot belong to the path-consistent $\mathcal{N}^*$. The proof for the invariant in line 8 is similar. □

Theorem 34 (Partial Correctness) *Algorithm PC-GEN is partially correct.*
Proof. By Lemma 33, it is sufficient to show that, when PC-GEN terminates, $\mathcal{N}$ is path-consistent. Assume that PC-GEN terminates with $\langle v, w \rangle \in C_{ij}$ such that $\neg PC_{ikj}(v, w)$. Let $u_1, \ldots, u_m$ be all the elements supporting $\langle v, w \rangle$ in the initial constraint network $\mathcal{N}_0$ (i.e. $C_{ik}(v, u_l) \wedge C_{kj}(u_l, w)$). At the end of PC-GEN, these supports have been deleted. We have $m > 0$, since otherwise $\langle v, w \rangle$ would have been removed from C_{ij} by line 2. Let u be the last support of $\langle v, w \rangle$ during the computation. Since we have $\neg PC_{ikj}(v, w)$ at the end of the execution, either $\langle v, u \rangle$ has been removed from C_{ik} or $\langle u, w \rangle$ has been removed from C_{kj}. Such a removal implied the insertion of $\langle i, k, j, v \rangle$ or $\langle j, k, i, u \rangle$ in the queue. As the algorithm is assumed to terminate, when this element will be dequeue and treated by LocalPathCons, $\langle v, w \rangle$ will be removed from C_{ij} (since $\neg PC_{ikj}(v, w)$) and thus $\langle v, w \rangle$ belongs to Δ_1. Contradiction. □

Lemma 35 (Termination) *In algorithm* PC-GEN*, if $s_1, \ldots, s_p$ are the numbers of new elements in Q after successive iterations of lines 5 or 12, then $s_1 + \ldots + s_p \leq O(n^3 d^2)$.*

Proof. Given that a tuple $\langle v, w\rangle$ can only be pruned at most once per constraint C_{ij} (specification of the subproblems), and given the specification of ENQUEUE, it follows that, for all $i, j, k \in arc(\mathcal{N})$, for all $v \in D$, the element $\langle i, k, j, v\rangle$ can be enqueued at most $O(d)$ times in the queue Q during the execution of PC-GEN. □

Theorem 36 *Algorithm PC-GEN terminates and is totally correct.*

*A.2. Correctness of PC-GEN**

The correctness of PC-GEN* is proved in three steps. We first show that in PC-GEN, if a tuple has the LH(m) property, then it is eventually removed. We then prove that, in an execution of PC-GEN, we may substitute executions of PATHCONS (or LOCALPATHCONS) by executions of PATHCONS* (or LOCALPATHCONS*). Hence the correctness of PC-GEN*. Let us first observe that the relaxed specifications does not influence Lemmas 33 and 35.

Lemma 37 *If, during the execution of PC-GEN, we have $\neg PC_{ikj}(v, w)$ and $\langle v, w\rangle$ LH(m) wrt ikj, for some v, w, i, k, j, m, then the tuple $\langle v, w\rangle$ will eventually be pruned from C_{ij}.*
Proof. The proof is by induction on m. For $m = 1$, we have $\langle i, k, j, v\rangle \in Q$ (the other case is similar). Termination ensures the existence of a call to LOCALPATHCONS(i, k, j, v). By hypothesis, we have $\neg PC_{ikj}(v, w)$. The tuple $\langle v, w\rangle$ will thus be in the resulting Δ set and pruned from C_{ij}.
For $m > 1$, we have $\neg PC_{ikj}(v, w)$ and

$$\exists u : \ \langle v, u\rangle \in C_{ik} \ \wedge \ \exists k' : \ \neg PC_{ik'k}(v, u) \ \wedge \ (\langle v, u\rangle \text{ is LH}(m-1) \text{ for } ik'k)$$

(the other case is similar). By induction hypothesis, the tuple $\langle v, u\rangle$ will eventually be pruned from C_{ik}, inducing the insertion of $\langle i, k, j, v\rangle$ in the queue. We are now in a similar case than for $m = 1$. □

Theorem 38 (Correctness of PC-GEN*) *Algorithm PC-GEN* is totally correct.*
Proof. Given that PC-GEN* always terminates and that the parametric procedures may now compute smaller Δ sets, it is sufficient to prove that all the postponed tuples will eventually be pruned. Let us consider an execution of PC-GEN*. Let p be the number of sets Δ computed by PATHCONS* and LOCALPATHCONS* which do not respect the initial specification of the parametric procedures. The proof is by induction on p. For $p = 0$, PC-GEN* is PC-GEN. For $p \geq 1$, consider the p^{th} call of these calls to PATHCONS* and LOCALPATHCONS*. Except for this call, the remaining part of the execution of PC-GEN* is now identical to an execution of PC-GEN. By Lemma 37, all the postponed tuples will eventually be pruned. The induction hypothesis can now by applied to the other $p - 1$ calls; the remaining postponed tuples will thus eventually be pruned. □

procedure PROPAGATE(**in** i, k, j, $\langle v, w\rangle$, BOUNDED, NEXT,
out Δ)

Let v_k = NEXT$^k(v)$,
w_k and Δ_k such that BOUNDED$(i, k, j, \langle v_k, w\rangle, \Delta_k, w_k)$,
$m = max\{k \mid \Delta_k \neq \emptyset \wedge w_k = w\}$.

Post: $\Delta = \bigcup_{1\leq k\leq m+1} \Delta_k$

begin
  $\Delta := \emptyset$;
  $v_{calc} := v$;
  repeat
    v_{calc} :=NEXT(v_{calc});
    BOUNDED(i, k, j, $\langle v_{calc}, w\rangle$, Δ_{calc}, w_{calc}) ;
    $\Delta := \Delta \cup \Delta_{calc}$;
  until ($w_{calc} \neq w$);
end

procedure BOUNDEDMIN(**in** i, k, j, $\langle v, w\rangle$, **out** Δ, w_{min})

Post: $w_{min} = max\{w \in D_j(C_{ji}) \mid \forall w' \in$ [MIN$(v, i, j), w]$:
EMPTYSUPPORT$(v, w', i, k, j)\}$
$\Delta = \{ \langle v, w'\rangle \mid w' \in$ [MIN$(v, i, j), w_{min}] \}$

begin
  $\Delta := \emptyset$;
  w_2 :=MIN(v, i, j);
  while $(w_2 \leq w) \wedge \neg$EMPTYSUPPORT(v,w_2,i,k,j) **do**
  begin
    $\Delta := \Delta \cup \{\langle v, w_2\rangle\}$;
    w_2 := SUCC(w_2);
  end ;
  w_{min} := PRED(w_2);
end

procedure BOUNDEDMAX(**in** i, k, j, $\langle v, w\rangle$, **out** Δ, w_{max})

Post: $w_{max} = min\{w \in D_j(C_{ji}) \mid \forall w' \in [w$,MAX$(v, i, j)]$:
EMPTYSUPPORT$(v, w', i, k, j)\}$
$\Delta = \{ \langle v, w'\rangle \mid w' \in [w_{max}$,MAX$(v, i, j)] \}$

begin
  $\Delta := \emptyset$;
  w_2 :=MAX(v, i, j);
  while $(w_2 \geq w) \wedge \neg$EMPTYSUPPORT(v,w_2,i,k,j) **do**
  begin
    $\Delta := \Delta \cup \{\langle v, w_2\rangle\}$;
    w_2 := PRED(w_2);
  end ;
  w_{max} := SUCC(w_2);
end