Constraint Propagation

Christian Bessiere
Technical Report LIRMM 06020
CNRS/University of Montpellier
March 2006

Constraint propagation is a form of inference, not search, and as such is more "satisfying", both technically and aesthetically.
—E.C. Freuder, 2005.

1 Introduction

Constraint reasoning involves various types of techniques to tackle the inherent intractability of the problem of satisfying a set of constraints. Constraint propagation is one of those types of techniques. Constraint propagation is central to the process of solving a constraint problem, and we could hardly think of constraint reasoning without it.

Constraint propagation is a very general concept that appears under different names depending on both periods and authors. Among these names, we can find constraint relaxation, filtering algorithms, narrowing algorithms, constraint inference, simplification algorithms, label inference, local consistency enforcing, rules iteration, chaotic iteration.

Constraint propagation embeds any reasoning which consists in explicitly forbidding values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise. For instance, in a crossword-puzzle, when you discard the words NORWAY and SWEDEN from the set of European countries that can fit a 6-digit slot because the second letter must be a 'R', you propagate a constraint. In a problem containing two variables x_1 and x_2 taking integer values in 1..10, and a constraint specifying that $|x_1 - x_2| > 5$, by propagating this constraint we can forbid values 5 and 6 for both x_1 and x_2. Explicating these 'nogoods' is a way to reduce the space of combinations that will be explored by a search mechanism.

The concept of constraint propagation can be found in other fields under different kinds and names. (See for instance the propagation of clauses by 'unit propagation' in propositional calculus [40].) Nevertheless, it is in constraint reasoning that this concept shows its most accomplished form. There is no other field in which the concept of constraint propagation appears in such a variety of forms, and in which its characteristics have been so deeply analyzed.

In the last 30 years, the scientific community has put a lot of effort in formalizing and characterizing this ubiquitous concept of constraint propagation and in proposing algorithms for propagating constraints. This formalization can be presented along two main lines: local consistencies and rules iteration. Local consistencies define properties that the constraint problem must satisfy *after* constraint propagation. This way, the operational behavior is left completely open, the only requirement being to achieve the given property on the output. The rules iteration approach, on the contrary, defines properties on the process of propagation itself, that is, properties on the kind/order of operations of reduction applied to the problem.

This report does not include *data-flow* constraints [76], even if this line of research has been the focus of quite a lot of work in interactive applications and if some of these papers speak about 'propagation' on these constraints [27]. They are indeed quite far from the techniques appearing in constraint programming.

The rest of this report is organized as follows. Section 2 contains basic definitions and notations used throughout the report. Section 3 formalizes all constraint propagation approaches within a unifying framework. Sections 4–9 contain the main existing types of constraint propagation. Each of these sections presents the basics on the type of propagation addressed and goes briefly into sharper or more recent advances on the subject.

2 Background

The notations used in this report have been chosen to support all notions presented. I tried to remain on the borderline between 'heavy abstruse notations' and 'ambiguous definitions', hoping I never fall too much on one side or the other of the edge.

A *constraint satisfaction problem (CSP)* involves finding solutions to a constraint network, that is, assignments of values to its variables that satisfy all its constraints. Constraints specify combinations of values that given subsets of variables are allowed to take. In this report, we are only concerned with constraint satisfaction problems where variables take their value in a *finite* domain. Without loss of generality, I assume these domains are mapped on the set $\mathbb{Z}$ of integers, and so, I consider only *integer* variables, that is, variables with a domain being a finite subset of $\mathbb{Z}$.

Definition 1 (Constraint). *A* constraint *c is a relation defined on a sequence of variables $X(c) = (x_{i_1}, \ldots, x_{i_{|X(c)|}})$, called the* scheme *of c. c is the subset of $\mathbb{Z}^{|X(c)|}$ that contains the combinations of values (or* tuples*) $\tau \in \mathbb{Z}^{|X(c)|}$ that* satisfy *c. $|X(c)|$ is called the* arity *of c. Testing whether a tuple τ satisfies a constraint c is called a* constraint check*.*

A constraint can be specified extensionally by the list of its satisfying tuples, or intensionally by a formula that is the characteristic function of the constraint. Definition 1 allows constraints with an *infinite* number of satisfying tuples. I sometimes write $c(x_1, \ldots, x_k)$ for a constraint c with scheme

$X(c) = (x_1, \dots, x_k)$. Constraints of arity 2 are called *binary* and constraints of arity greater than 2 are called *non-binary*. *Global* constraints are classes of constraints defined by a formula of arbitrary arity (see Section 9.2).

Example 2. The constraint $\texttt{alldifferent}(x_1, x_2, x_3) \equiv (v_i \neq v_j \wedge v_i \neq v_k \wedge v_j \neq v_k)$ allows the infinite set of 3-tuples in $\mathbb{Z}^3$ such that all values are different. The constraint $c(x_1, x_2, x_3) = \{(2, 2, 3), (2, 3, 2), (2, 3, 3), (3, 2, 2), (3, 2, 3), (3, 3, 2)\}$ allows the finite set of 3-tuples containing both values 2 and 3 and only them.

Definition 3 (Constraint network). *A* constraint network *(or* network*) is composed of:*

- *a finite sequence of integer variables $X = (x_1, \dots, x_n)$,*
- *a domain for X, that is, a set $D = D(x_1) \times \ldots \times D(x_n)$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values, given in extension,[1] that variable x_i can take, and*
- *a set of constraints $C = \{c_1, \dots, c_e\}$, where variables in $X(c_j)$ are in X.*

Given a network N, I sometimes use X_N, D_N and C_N to denote its sequence of variables, its domain and its set of constraints. Given a variable x_i and its domain $D(x_i)$, $min_D(x_i)$ denotes the smallest value in $D(x_i)$ and $max_D(x_i)$ its greatest one. (Remember that we consider integer variables.)

In the whole report, I consider constraints involving at least two variables. This is not a restriction because domains of variables are semantically equivalent to unary constraints. They are separately specified in the definition of constraint network because the domains are given extensionally whereas a constraint c can be defined by any Boolean function on $\mathbb{Z}^{|X(c)|}$ (in extension or not). I also consider that no variable is repeated in the scheme of a constraint. This restriction could be relaxed in most cases, but it simplifies the notations. The vocabulary of graphs is often used to describe networks. A network can indeed be associated with a *(hyper)graph* where variables are nodes and where schemes of constraints are (hyper)edges.

According to Definitions 1 and 3, the variables X_N of a network N and the scheme $X(c)$ of a constraint $c \in C_N$ are sequences of variables, not sets. This is required because the order of the values matters for tuples in D_N or in c. Nevertheless, it simplifies a lot the notations to consider sequences as sets when no confusion is possible. For instance, given two constraints c and c', $X(c) \subseteq X(c')$ means that constraint c involves only variables that are in the scheme of c', whatever their ordering in the scheme. Given a tuple τ on a sequence Y of variables, and given a sequence $W \subseteq Y$, $\tau[W]$ denotes the restriction of τ to the variables in W, ordered according to W. Given $x_i \in Y, \tau[x_i]$ denotes the value of x_i in τ. If $X(c) = X(c')$, $c \subseteq c'$ means that for all $\tau \in c$ the reordering of τ according to $X(c')$ satisfies c'.

[1] The condition on the domains given in extension can be relaxed, especially in numerical problems where variables take values in a discretization of the reals. (See Chapter 16 in Part II of [109].)

Example 4. Let $(1, 1, 2, 4, 5)$ be a tuple on $Y = (x_1, x_2, x_3, x_4, x_5)$ and $W = (x_3, x_2, x_4)$. $\tau[x_3]$ is the value 2 and $\tau[W]$ is the tuple $(2, 1, 4)$. Given $c(x_1, x_2, x_3)$ defined by $x_1 + x_2 = x_3$ and $c'(x_2, x_1, x_3)$ defined by $x_2 + x_1 \leq x_3$, we have $c \subseteq c'$.

We also need the concepts of projection, intersection, union and join. Given a constraint c and a sequence $Y \subseteq X(c)$, $\pi_Y(c)$ denotes the *projection* of c on Y, that is, the relation with scheme Y that contains the tuples that can be extended to a tuple on $X(c)$ satisfying c. Given two constraints c_1 and c_2 sharing the same scheme $X(c_1) = X(c_2)$, $c_1 \cap c_2$ (resp. $c_1 \cup c_2$) denotes the *intersection* (resp. the *union*) of c_1 and c_2, that is, the relation with scheme $X(c_1)$ that contains the tuples τ satisfying both c_1 and c_2 (resp. satisfying c_1 or c_2). Given a set of constraints $\{c_1, \ldots, c_k\}$, $\bowtie_{j=1}^{k} c_j$ (or $\bowtie \{c_1, \ldots, c_k\}$) denotes the *join* of $c_1, \ldots, c_k$, that is, the relation with scheme $\cup_{j=1}^{k} X(c_j)$ that contains the tuples τ such that $\tau[X(c_j)] \in c_j$ for all $j, 1 \leq j \leq k$.

Backtracking algorithms are based on the principle of assigning values to variables until all variables are *instantiated*.

Definition 5 (Instantiation). *Given a network $N = (X, D, C)$,*

- *An* instantiation *I on $Y = (x_1, \ldots, x_k) \subseteq X$ is an assignment of values $v_1, \ldots v_k$ to the variables $x_1, \ldots, x_k$, that is, I is a tuple on Y. I can be denoted by $((x_1, v_1), \ldots, (x_k, v_k))$ where (x_i, v_i) denotes the value v_i for x_i.*
- *An instantiation I on Y is* valid *if for all $x_i \in Y, I[x_i] \in D(x_i)$.*
- *An instantiation I on Y is* locally consistent *iff it is valid and for all $c \in C$ with $X(c) \subseteq Y$, $I[X(c)]$ satisfies c. If I is not locally consistent, it is* locally inconsistent.
- *A* solution *to a network N is an instantiation I on X which is locally consistent. The set of solutions of N is denoted by $sol(N)$.*
- *An instantiation I on Y is* globally consistent *(or* consistent*) if it can be extended to a solution (i.e., there exists $s \in sol(N)$ with $I = s[Y]$).*

Example 6. Let $N = (X, D, C)$ be a network with $X = (x_1, x_2, x_3, x_4)$, $D(x_i) = \{1, 2, 3, 4, 5\}$ for all $i \in [1..4]$ and $C = \{c_1(x_1, x_2, x_3), c_2(x_1, x_2, x_3), c_3(x_2, x_4)\}$ with $c_1(x_1, x_2, x_3) =$ `alldifferent`(x_1, x_2, x_3), $c_2(x_1, x_2, x_3) \equiv (x_1 \leq x_2 \leq x_3)$, and $c_3(x_2, x_4) \equiv (x_4 \geq 2 \cdot x_2)$. We thus have $\pi_{\{x_1, x_2\}}(c_1) \equiv (x_1 \neq x_2)$ and $c_1 \cap c_2 \equiv (x_1 < x_2 < x_3)$. $I_1 = ((x_1, 1), (x_2, 2), (x_4, 7))$ is a non valid instantiation on $Y = (x_1, x_2, x_4)$ because $7 \notin D(x_4)$. $I_2 = ((x_1, 1), (x_2, 1), (x_4, 3))$ is a locally consistent instantiation on Y because c_3 is the only constraint with scheme included in Y and it is satisfied by $I_2[X(c_3)]$. However, I_2 is not globally consistent because it does not extend to a solution of N. $sol(N) = \{(1, 2, 3, 4), (1, 2, 3, 5)\}$.

There are many works in the constraint reasoning community that put some restrictions on the definition of a constraint network. These restrictions can

have some consequences on the notions handled. I define the main restrictions appearing in the literature and that I will use later.

Definition 7 (Normalized and binary networks).

- *A network N is* normalized *iff two different constraints in C_N do not involve exactly the same variables.*
- *A network N is* binary *iff for all $c_i \in C_N, |X(c_i)| = 2$.*

When a network is both binary and normalized, a constraint $c(x_i, x_j) \in C$ is often denoted by c_{ij}. To simplify even further the notations, c_{ji} denotes its *transposition*, i.e., the constraint $c(x_j, x_i) = \{(v_j, v_i) \mid (v_i, v_j) \in c_{ij}\}$, and since there cannot be ambiguity with another constraint, I act as if c_{ji} was in C as well.

Given two normalized networks $N = (X, D, C)$ and $N' = (X, D', C')$, $N \sqcup_N N'$ denotes the network $N'' = (X, D'', C'')$ with $D'' = D \cup D'$ and $C'' = \{c'' \mid \exists c \in C, \exists c' \in C', X(c) = X(c') \text{ and } c'' = c \cup c'\}$.

The constraint reasoning community often used constraints with a finite number of tuples, and even more, constraints that only allow valid tuples, that is, combinations of values from the domains of the variables involved. I call these constraints 'embedded'.

Definition 8 (Embedded network). *Given a network N and a constraint $c \in C_N$, the* embedding *of c in D_N is the constraint $\hat{c}$ with scheme $X(c)$ such that $\hat{c} = c \cap \pi_{X(c)}(D_N)$. A network N is* embedded *iff for all $c \in C_N, c = \hat{c}$.*

In complexity analysis, we sometimes need to refer to the size of a network. The *size* of a network N is equal to $|X_N| + \sum_{x_i \in X_N} |D_N(x_i)| + \sum_{c_j \in C_N} \|c_j\|$, where $\|c\|$ is equal to $|X(c)| \cdot |c|$ if c is given in extension, or equal to the size of its encoding if c is defined by a Boolean function.

3 Formal Viewpoint

This section formally characterizes the concept of constraint propagation. The aim is essentially to relate the different notions of constraint propagation.

The constraint satisfaction problem being NP-complete, it is usually solved by backtrack search procedures that try to extend a partial instantiation to a global one that is consistent. Exploring the whole space of instantiations is of course too expensive. The idea behind constraint propagation is to make the constraint network more explicit (or *tighter*) so that backtrack search commits into less inconsistent instantiations by detecting local inconsistency earlier. I first introduce the following preorder on constraint networks.

Definition 9 (Preorder $\preceq$ on networks). *Given two networks N and N', we say that $N' \preceq N$ iff $X_{N'} = X_N$ and any instantiation I on $Y \subseteq X_N$ locally inconsistent in N is locally inconsistent in N' as well.*

From the definition of local inconsistency of an instantiation (Definition 5) I derive the following property of constraint networks ordered according to $\preceq$.

Proposition 10. *Given two networks N and N', $N' \preceq N$ iff $X_{N'} = X_N$, $D_{N'} \subseteq D_N$,[2] and for any constraint $c \in C_N$, for any tuple τ on $X(c)$ that does not satisfy c, either τ is not valid in $D_{N'}$ or there exists a constraint c' in $C_{N'}$, $X(c') \subseteq X(c)$, such that $\tau[X(c')] \notin c'$.*

The relation $\preceq$ is not an order because there can be two different networks N and N' with $N \preceq N' \preceq N$.

Definition 11 (Nogood-equivalence). *Two networks N and N' such that $N \preceq N' \preceq N$ are said to be* nogood-equivalent*. (A* nogood *is a partial instantiation that does not lead to a solution.)*

Example 12. Let $N = (X, D, C)$ be the network with $X = \{x_1, x_2, x_3\}$, $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3, 4\}$ and $C = \{x_1 < x_2, x_2 < x_3,\ c(x_1, x_2, x_3)\}$ where $c(x_1, x_2, x_3) = \{(111), (123), (222), (333)\}$. Let $N' = (X, D, C')$ be the network with $C' = \{x_1 < x_2, x_2 < x_3,\ c'(x_1, x_2, x_3)\}$, where $c'(x_1, x_2, x_3) = \{(123), (231), (312)\}$. The only difference between N and N' is that the latter contains c' instead of c. For any tuple τ on $X(c)$ (resp. $X(c')$) that does not satisfy c (resp. c'), there exists a constraint in C' (resp. in C) that makes τ locally inconsistent. As a result, $N \preceq N' \preceq N$ and N and N' are nogood-equivalent.

Constraint propagation transforms a network N by tightening D_N, by tightening constraints from C_N, or by adding new constraints to C_N. Constraint propagation does not remove redundant constraints, which is more a reformulation task. I define the space of networks that can be obtained by constraint propagation on a network N.

Definition 13 (Tightenings of a network). *The space $\mathcal{P}_N$ of all possible* tightenings *of a network $N = (X, D, C)$ is the set of networks $N' = (X, D', C')$ such that $D' \subseteq D$ and for all $c \in C$ there exists $c' \in C'$ with $X(c') = X(c)$ and $c' \subseteq c$.*

Note that $\mathcal{P}_N$ does not contain all networks $N' \preceq N$. In Example 12, $N' \notin \mathcal{P}_N$ because $c' \not\subseteq c$. However, if $N'' = (X, D, C'')$ with $C'' = \{x_1 < x_2, x_2 < x_3, c'' = c \cup c'\}$, we have $N \in \mathcal{P}_{N''}$ and $N' \in \mathcal{P}_{N''}$. The set of networks $\mathcal{P}_N$ together with $\preceq$ forms a preordered set. The top element of $\mathcal{P}_N$ according to $\preceq$ is N itself and the bottom elements are the networks with empty domains.[3] In $\mathcal{P}_N$ we are particularly interested in networks that preserve the set of solutions of N. $\mathcal{P}_N^{sol}$ denotes the subset of $\mathcal{P}_N$ containing only the elements N' of $\mathcal{P}_N$ such that $sol(N') = sol(N)$. Among the networks in $\mathcal{P}_N^{sol}$, those that are the smallest according to $\preceq$ have interesting properties.

[2] $D_{N'} \subseteq D_N$ because we supposed that networks do not contain unary constraints, and so, instantiations of size 1 can be made locally inconsistent only because of the domains.

[3] Remember that we consider that unary constraints are expressed in the domains.

Proposition 14 (Global consistency). *Let* $N = (X, D, C)$ *be a network, and* $G_N = (X, D_G, C_G)$ *be a network in* $\mathcal{P}_N^{sol}$*. If* G_N *is such that for all* $N' \in \mathcal{P}_N^{sol}, G_N \preceq N'$*, then any instantiation* I *on* $Y \subseteq X$ *which is locally consistent in* G_N *can be extended to a solution of* N*.* G_N *is called a* globally consistent *network.*

Proof. Suppose there exists an instantiation I on $Y \subseteq X$ locally consistent in G_N which does not extend to a solution. Build the network $N' = (X, D_G, C_G \cup \{c\})$ where $X(c) = Y$ and $c = \mathbb{Z}^{|Y|} \setminus \{I\}$. $N' \in \mathcal{P}_N^{sol}$ because $G_N \in \mathcal{P}_N^{sol}$ and I does not extend to a solution of N. In addition, I is locally inconsistent in N'. So, $G_N \not\preceq N'$. □

Thanks to Proposition 14 we see the advantage of having a globally consistent network of N. A simple brute-force backtrack search procedure applied on a globally consistent network is guaranteed to produce a solution in a backtrack-free manner. However, globally consistent networks have a number of disadvantages that make them impossible to use in practice. A globally consistent network is not only exponential in time to compute, but in addition, its size is in general exponential in the size of N. In fact, building a globally consistent network is similar to generating and storing all minimal nogoods of N. Building a globally consistent network is so hard that a long tradition in constraint programming is to try to transform N into an element of $\mathcal{P}_N^{sol}$ as close as possible to global consistency at reasonable cost (usually keeping polynomial time and space). This is constraint propagation.

Rules iteration and local consistencies are two ways of formalizing constraint propagation. *Rules iteration* consists in characterizing for each constraint (or set of constraints) a set of reduction rules that tighten the network. *Reduction rules* are sufficient conditions to rule out values (or instantiations) that have no chance to appear in a solution. The second —and most well-known— way of considering constraint propagation is via the notion of local consistency. A *local consistency* is a property that characterizes some necessary conditions on values (or instantiations) to belong to solutions. A local consistency property (denoted by Φ) is defined regardless of the domains or constraints that will be present in the network. A network is Φ-consistent if and only if it satisfies the property Φ.

It is difficult to say more about constraint propagation in completely general terms. The preorder $(\mathcal{P}_N, \preceq)$ is indeed too weak to characterize the features of constraint propagation. Most of the constraint propagation techniques appearing in constraint programming (or at least those that are used in solvers) are limited to modifications of the domains. So, I first concentrate on this subcase, that I call *domain-based* constraint propagation. I will come back to the general case in Section 5.

Definition 15 (Domain-based tightenings). *The space* $\mathcal{P}_{ND}$ *of* domain-based tightenings *of a network* $N = (X, D, C)$ *is the set of networks in* $\mathcal{P}_N$ *with the same constraints as* N*, that is,* $N' \in \mathcal{P}_{ND}$ *iff* $X_{N'} = X$*,* $D_{N'} \subseteq D$ *and* $C_{N'} = C$*.*

Proposition 16 (Partial order on networks). *Given a network N, the relation $\preceq$ restricted to the set $\mathcal{P}_{ND}$ is a partial order (denoted by $\leq$).*

$(\mathcal{P}_{ND}, \leq)$ is a partially ordered set (*poset*) because given two networks $N_1 = (X_1, D_1, C_1)$ and $N_2 = (X_2, D_2, C_2)$, $N_1 \leq N_2 \leq N_1$ implies that $X_1 = X_2$, $C_1 = C_2$, and $D_1 \subseteq D_2 \subseteq D_1$, which means that $N_1 = N_2$. In fact, the poset $(\mathcal{P}_{ND}, \leq)$ is isomorphic to the partial order $\subseteq$ on D_N. We are interested in the subset $\mathcal{P}^{sol}_{ND}$ of $\mathcal{P}_{ND}$ containing all the networks that preserve the set of solutions of N. $\mathcal{P}^{sol}_{ND}$ has the same top element as $\mathcal{P}_{ND}$, namely N itself, and a *unique* bottom element $G_{ND} = (X_N, D_G, C_N)$, where for any $x_i \in X_N$, $D_G(x_i)$ only contains values belonging to a solution of N, i.e., $D_G(x_i) = \pi_{\{x_i\}}(sol(N))$. Such a network was named *variable-completable* by Freuder [56].

Domain-based constraint propagation looks for an element in $\mathcal{P}^{sol}_{ND}$ on which the search space to explore is smaller (that is, values have been pruned from the domains). Since finding G_{ND} is NP-hard (consistency of N reduces to checking non emptiness of domains in G_{ND}), domain-based constraint propagation usually consists of polynomial techniques that produce a network which is an approximation of G_{ND}. The network N' produced by a domain-based constraint propagation technique always verifies $G_{ND} \leq N' \leq N$, that is, $D_G \subseteq D_{N'} \subseteq D_N$.

Domain-based rules iteration consists in applying for each constraint $c \in C_N$ a set of reduction rules that rule out values of x_i that cannot appear in a tuple satisfying c. Domain-based reduction rules are also named *propagators*. For instance, if $c \equiv (|x_1 - x_2| = k)$, a propagator for c on x_1 can be $D_N(x_1) \leftarrow D_N(x_1) \cap [min_{D_N}(x_2) - k \,..\, min_{D_N}(x_2) + k]$. Applying propagators iteratively tightens D_N while preserving the set of solutions of N. In other words, propagators slide down the poset $(\mathcal{P}_{ND}, \leq)$ without moving out of $\mathcal{P}^{sol}_{ND}$. Reduction rules will be presented in Section 8. From now on, we concentrate on domain-based local consistencies. Any property Φ that specifies a necessary condition on values to belong to solutions can be considered as a domain-based local consistency. Nevertheless, we usually consider only those properties that are stable under union.

Definition 17 (Stability under union). *A domain-based property Φ is* stable under union *iff for any Φ-consistent networks $N_1 = (X, D_1, C)$ and $N_2 = (X, D_2, C)$, the network $N' = (X, D_1 \cup D_2, C)$ is Φ-consistent.*

Example 18. Let Φ be the property that guarantees that for each constraint c and variable $x_i \in X(c)$, at least half of the values in $D(x_i)$ belong to a valid tuple satisfying c. Let $X = (x_1, x_2)$ and $C = \{x_1 = x_2\}$. Let D_1 be the domain with $D_1(x_1) = \{1, 2\}$ and $D_1(x_2) = \{2\}$. Let D_2 be the domain with $D_2(x_1) = \{2, 3\}$ and $D_2(x_2) = \{2\}$. (X, D_1, C) and (X, D_2, C) are both Φ-consistent but $(X, D_1 \cup D_2, C)$ is not Φ-consistent because among the three values for x_1, only value 2 can satisfy the constraint $x_1 = x_2$. Φ is not stable under union.

Stability under union brings very useful features for local consistencies.

Among all networks in $\mathcal{P}_{ND}$ that verify a local consistency Φ, there is a particular one.

Theorem 19 (Φ-closure). *Let $N = (X, D, C)$ be a network and Φ be a domain-based local consistency. Let $\Phi(N)$ be the network (X, D_Φ, C) where $D_\Phi = \cup\{D' \subseteq D \mid (X, D', C)$ is Φ-consistent$\}$. If Φ is stable under union, $\Phi(N)$ is Φ-consistent and is the* unique *network in $\mathcal{P}_{ND}$ such that for any Φ-consistent network $N' \in \mathcal{P}_{ND}$, $N' \leq \Phi(N)$. $\Phi(N)$ is called the* Φ-closure *of N. (By convention, we suppose $(X, \emptyset, C)$ is Φ-consistent.)*

$\Phi(N)$ has some interesting properties. The first one I can point out is that it preserves the solutions: $sol(\Phi(N)) = sol(N)$. This is not the case for all Φ-consistent networks in $\mathcal{P}_{ND}$.

Example 20. Let Φ be the property that guarantees that all values for all variables can be extended consistently to a second variable. Consider the network $N = (X, D, C)$ with variables x_1, x_2, x_3, domains all equal to $\{1, 2\}$ and $C = \{x_1 \leq x_2, x_2 \leq x_3, x_1 \neq x_3\}$. Let D_1 be the domain with $D_1(x_1) = D_1(x_2) = \{1\}$ and $D_1(x_3) = \{2\}$. (X, D_1, C) is Φ-consistent but does not contain the solution $(x_1 = 1, x_2 = 2, x_3 = 2)$ which is in $sol(N)$. In fact, $\Phi(N) = (X, D_\Phi, C)$ with $D_\Phi(x_1) = \{1\}$, $D_\Phi(x_2) = \{1, 2\}$ and $D_\Phi(x_3) = \{2\}$.

Computing a particular Φ-consistent network of $\mathcal{P}_{ND}$ can be difficult. G_{ND} for instance, is obviously Φ-consistent for any domain-based local consistency Φ, but it is NP-hard to compute. The second interesting property of $\Phi(N)$ is that it can be computed by a greedy algorithm.

Proposition 21 (Fixpoint). *If a domain-based consistency property Φ is stable under union, then for any network $N = (X, D, C)$, the network $N' = (X, D', C)$, where D' is obtained by iteratively removing values that do not satisfy Φ until no such value exists, is the Φ-closure of N.*

Corollary 22. *If a domain-based consistency property Φ is polynomial to check, finding $\Phi(N)$ is polynomial as well.*

By *achieving* (or *enforcing*) Φ-consistency on a network N, I mean finding the Φ-closure $\Phi(N)$.

I define a partial order on local consistencies to express how much they permit to go down the poset $(\mathcal{P}_{ND}, \leq)$. A domain-based local consistency Φ_1 is *at least as strong as* another local consistency Φ_2 if and only if for any network N, $\Phi_1(N) \leq \Phi_2(N)$. If in addition there exists a network N' such that $\Phi_1(N') < \Phi_2(N')$, then Φ_1 is *strictly stronger* than Φ_2. If there exist networks N' and N'' such that $\Phi_1(N') < \Phi_2(N')$ and $\Phi_2(N'') < \Phi_1(N'')$, Φ_1 and Φ_2 are *incomparable*.

When networks are both normalized and embedded, stability under union, Φ-closure, and the 'stronger' relation between local consistencies can be extended to local consistencies other than domain-based ones by simply replacing $\mathcal{P}_{ND}$ by $\mathcal{P}_N$, the union on domains $\cup$ by the union on networks $\sqcup_N$ (see Section 2), and the partial order $\leq$ on $\mathcal{P}_{ND}$ by the preorder $\preceq$ on $\mathcal{P}_N$ (see Section 5).

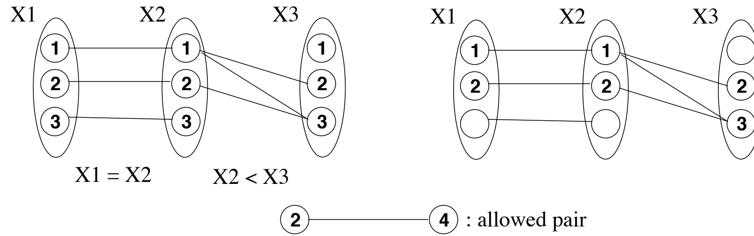

Figure 1: Network of Example 23 before arc consistency (left) and after (right).

4 Arc Consistency

Arc consistency is the oldest and most well-known way of propagating constraints. This is indeed a very simple and natural concept that guarantees every value in a domain to be consistent with every constraint.

Example 23. Let N be the network depicted in Fig. 1(left). It involves three variables x_1, x_2 and x_3, domains $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$, and constraints $c_{12} \equiv (x_1 = x_2)$ and $c_{23} \equiv (x_2 < x_3)$. N is not arc consistent because there are some values inconsistent with some constraints. Checking constraint c_{12} does not permit to remove any value. But when checking constraint c_{23}, we see that $(x_2, 3)$ must be removed because there is no value greater than it in $D(x_3)$. We can also remove value 1 from $D(x_3)$ because of constraint c_{23}. Removing 3 from $D(x_2)$ causes in turn the removal of value 3 for x_1 because of constraint c_{12}. Now, all remaining values are compatible with all constraints.

REF-ARF [51], is probably one of the first systems incorporating a feature which looks similar to arc consistency (even if the informal description does not permit to be sure of the equivalence). In papers by Waltz [125] and Gaschnig [61], the correspondence is more evident since algorithms for achieving arc consistency were presented. But the seminal papers on the subject are due to Mackworth, who is the first who clearly defined the concept of arc consistency for binary constraints [86], who extended definitions and algorithms to non-binary constraints [88], and who analyzed the complexity [89].

I give a definition of arc consistency in its most general form, i.e., for arbitrary constraint networks (in which case it is often called generalized arc consistency). In its first formal presentation, Mackworth limited the definition to binary normalized networks.

Definition 24 ((Generalized) arc consistency ((G)AC)). *Given a network $N = (X, D, C)$, a constraint $c \in C$, and a variable $x_i \in X(c)$,*

- *A value $v_i \in D(x_i)$ is* consistent with c *in D iff there exists a valid tuple τ satisfying c such that $v_i = \tau[\{x_i\}]$. Such a tuple is called a* support *for (x_i, v_i) on c.*

- *The domain D is* (generalized) arc consistent *on c for x_i iff all the values in $D(x_i)$ are consistent with c in D (that is, $D(x_i) \subseteq \pi_{\{x_i\}}(c \cap \pi_{X(c)}(D))$).*
- *The network N is* (generalized) arc consistent *iff D is (generalized) arc consistent for all variables in X on all constraints in C.*
- *The network N is* arc inconsistent *iff $\emptyset$ is the only domain tighter than D which is (generalized) arc consistent for all variables on all constraints.*

By notation abuse, when there is no ambiguity on the domain D to consider, we often say 'constraint c is arc consistent' instead of 'D is arc consistent on c for all $x_i \in X(c)$'. We also say 'variable x_i is arc consistent on constraint c' instead of 'all values in $D(x_i)$ are consistent with c in D'. When a constraint c_{ij} is binary and a tuple $\tau = (v_i, v_j)$ supports (x_i, v_i) on c_{ij}, we often refer to (x_j, v_j) (rather than to τ itself) when we speak about a 'support for (x_i, v_i)'.

Historically, many papers on constraint satisfaction made the simplifying assumption that networks are binary and normalized. This has the advantage that notations become much simpler (see Section 2) and new concepts are easier to present. But this had some strange effects that we must bear in mind.

First, the name 'arc consistency' is so strongly bound to binary networks that even if the definition is perfectly the same for both binary and non-binary constraints, a different name has often been used for arc consistency on non-binary constraints. Some papers use *hyper* arc consistency, or *domain* consistency, but the most common name is *generalized* arc consistency. In the following, I will use indifferently arc consistency (AC) or generalized arc consistency (GAC), though I will use GAC when the network is explicitly non-binary.

The second strange effect of associating AC with binary normalized networks is the confusion between the notions of arc consistency and 2-consistency. (As we will see in Section 5, 2-consistency guarantees that any instantiation of a value to a variable can be consistently extended to any second variable.) On binary networks, 2-consistency is at least as strong as AC. When the binary network is normalized, arc consistency and 2-consistency are equivalent. However, this is not true in general. The following examples show that 2-consistency is strictly stronger than AC on non normalized binary networks and that generalized arc consistency and 2-consistency are incomparable on arbitrary networks.

Example 25. Let N be a network involving two variables x_1 and x_2, with domains $\{1, 2, 3\}$, and the constraints $x_1 \leq x_2$ and $x_1 \neq x_2$. This network is arc consistent because every value has a support on every constraint. However, this network is not 2-consistent because the instantiation $x_1 = 3$ cannot be extended to x_2 and the instantiation $x_2 = 1$ cannot be extended to x_1.

Let N be a network involving three variables x_1, x_2, and x_3, with domains $D(x_1) = D(x_2) = \{2, 3\}$ and $D(x_3) = \{1, 2, 3, 4\}$, and the constraint `alldifferent`(x_1, x_2, x_3). N is 2-consistent because every value for any variable can be extended to a locally consistent instantiation on any second variable. However, this network is not GAC because the values 2 and 3 for x_3 do not have support on the `alldifferent` constraint.

4.1 Complexity of arc consistency

There are a number of questions related to GAC reasoning. It is worth analyzing their complexity. Bessiere et al. have characterized five questions that can be asked about a constraint [21]. Some of the questions are more of an academic nature whereas others are at the heart of propagation algorithms. These questions can be asked in general, or on a particular class of constraints, such as a given global constraint (see Section 9.2). These questions can be adapted to other local consistencies that we will present in latter sections. In the following, I use the notation Problem[data] to refer to the instance of Problem with the input 'data'.

> GACSupport
> **Instance.** A constraint c, a domain D on $X(c)$, and a value v for variable x_i in $X(c)$
> **Question.** Does value v for x_i have a support on c in D?

GACSupport is at the core of all generic arc consistency algorithms. GACSupport is generally asked for all values one by one.

> IsItGAC
> **Instance.** A constraint c, a domain D on $X(c)$
> **Question.** Does GACSupport$[c, D, x_i, v]$ answer 'yes' for each variable $x_i \in X(c)$ and each value $v \in D(x_i)$?

IsItGAC has both practical and theoretical importance. If enforcing GAC on a particular constraint is expensive, we may first test whether it is necessary or not to launch the propagation algorithm (i.e., whether the constraint is already GAC). On the academic side, this question is commonly used to compare different levels of local consistency.

> NoGACWipeOut
> **Instance.** A constraint c, a domain D on $X(c)$
> **Question.** Is there a non empty $D' \subseteq D$ on which IsItGAC$[c, D']$ answers 'yes'?

NoGACWipeOut occurs when GAC is maintained during search by a backtrack procedure. At each node in the search tree (i.e., after each instantiation of a value to a variable), we want to know if the remaining network can be made GAC without wiping out the domain. If not, we must unassign one of the variables already instantiated.

> maxGAC
> **Instance.** A constraint c, a domain D_0 on $X(c)$, and a domain $D \subseteq D_0$
> **Question.** Is $(X(c), D, \{c\})$ the arc consistent closure of $(X(c), D_0, \{c\})$?

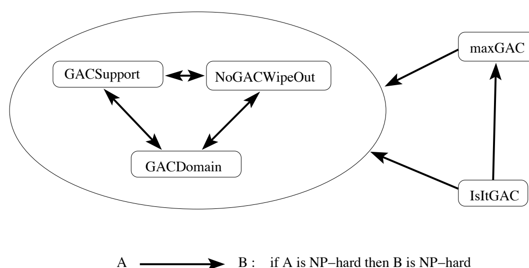

Figure 2: Dependencies between intractability of arc consistency questions

Arc consistency algorithms (see next subsection) are asked to return the arc consistent closure of a network, that is, the subdomain that is GAC and any larger subdomain is not GAC. MAXGAC characterizes this 'maximality' problem.

GACDOMAIN
Instance. A constraint c, a domain D_0 on $X(c)$
Output. The domain D such that MAXGAC$[c, D_0, D]$ answers 'yes'

GACDOMAIN returns the arc consistent closure, that is, the domain that a GAC algorithm computes. GACDOMAIN is not a decision problem as it computes something other than 'yes' or 'no'.

In [20, 21], Bessiere et al. showed that all five questions are NP-hard in general. In addition, they showed that on any particular class of constraints, NP-hardness of a question implies NP-hardness of other questions.

Theorem 26 (Dependencies in the NP-hardness of GAC questions). *Given a class $\mathcal{C}$ of constraints,* GACSUPPORT *is NP-hard on $\mathcal{C}$ iff* NOGACWIPEOUT *is NP-hard on $\mathcal{C}$.* GACSUPPORT *is NP-hard on $\mathcal{C}$ iff* GACDOMAIN *is NP-hard on $\mathcal{C}$. If* MAXGAC *is NP-hard on $\mathcal{C}$ then* GACSUPPORT *is NP-hard on $\mathcal{C}$. If* ISITGAC *is NP-hard on $\mathcal{C}$ then* MAXGAC *is NP-hard on $\mathcal{C}$.*

A summary of the dependencies in Theorem 26 is given in Fig. 2. Note that because each arrow from question A to question B in Fig. 2 means that A can be rewritten as a polynomial number of calls to B, we immediately derive that tractability of B implies tractability of A. Whereas the decision problems GACSUPPORT, ISITGAC, and NOGACWIPEOUT are in NP, MAXGAC may be outside NP. In fact, MAXGAC is D^P-complete in general. The D^P complexity class contains problems which are the conjunction of a problem in NP and one in coNP [101].

Assuming P $\neq$ NP, GAC reasoning is thus not tractable in general. In fact, the best complexity that can be achieved for an algorithm enforcing GAC on a network with any kind of constraints is in $O(erd^r)$, where e is the number of constraints and r is the largest arity of a constraint.

Though not related to GAC, constraint *entailment* ([70]) is a sixth question that is used by constraint solvers to speed up propagation. An entailed constraint can safely be disconnected from the network.

> ENTAILED
> **Instance.** A constraint c, a domain D on $X(c)$
> **Question.** Does ISITGAC$[c, D']$ answer 'yes' for all $D' \subseteq D$?

Entailment of c on D means that $D \subseteq c$. ENTAILED is coNP-complete in general. There is no dependency between intractability of entailment and intractability of the GAC questions. On a class $\mathcal{C}$ of constraints, ENTAILED can be tractable and the GAC questions intractable, or the reverse, or both tractable or intractable.

4.2 Arc consistency algorithms

Proposing efficient algorithms for enforcing arc consistency has always been considered as a central question in the constraint reasoning community. A first reason is that arc consistency is the basic propagation mechanism that is probably used in all solvers. A second reason is that the new ideas that permit to improve efficiency of arc consistency can usually be applied to algorithms achieving other local consistencies. This is why I spend some time presenting the main algorithms that have been introduced, knowing that the techniques involved can be used for other local consistencies presented in forthcoming sections. I follow a chronological presentation to emphasize the incremental process that led to the current algorithms.

4.2.1 AC3

The most well-known algorithm for arc consistency is the one proposed by Mackworth in [86] under the name AC3. It was proposed for binary normalized networks and actually achieves 2-consistency. It was extended to GAC in arbitrary networks in [88]. This algorithm is quite simple to understand. The burden of the general notations being not so high, I present it in its general version. (See Algorithm 1.)

The main component of GAC3 is the revision of an arc, that is, the update of a domain wrt a constraint.[4] Updating a domain $D(x_i)$ wrt a constraint c means removing every value in $D(x_i)$ that is not consistent with c. The function `Revise`(x_i, c) takes each value v_i in $D(x_i)$ in turn (line 2), and explores the space $\pi_{X(c)\setminus\{x_i\}}(D)$, looking for a support on c for v_i (line 3). If such a support is not found, v_i is removed from $D(x_i)$ and the fact that $D(x_i)$ has been changed is flagged (lines 4–5). The function returns true if the domain $D(x_i)$ has been reduced, false otherwise (line 6).

The main algorithm is a simple loop that revises the arcs until no change occurs, to ensure that all domains are consistent with all constraints. To avoid too many useless calls to `Revise` (as this is the case in the very basic AC

[4] The word 'arc' comes from the binary case but we also use it on non-binary constraints.

Algorithm 1: AC3 / GAC3

function `Revise3`**(in** x_i: variable; c: constraint**): Boolean** ;
begin
1 CHANGE $\leftarrow$ **false**;
2 **foreach** $v_i \in D(x_i)$ **do**
3 **if** $\nexists \tau \in c \cap \pi_{X(c)}(D)$ *with* $\tau[x_i] = v_i$ **then**
4 remove v_i from $D(x_i)$;
5 CHANGE $\leftarrow$ **true**;
6 return CHANGE ;
end

function `AC3/GAC3`**(in** X: set**): Boolean** ;
begin
/* initalisation */;
7 $Q \leftarrow \{(x_i, c) \mid c \in C, x_i \in X(c)\}$;
/* propagation */;
8 **while** $Q \neq \emptyset$ **do**
9 select and remove (x_i, c) from Q;
10 **if** *Revise*(x_i, c) **then**
11 **if** $D(x_i) = \emptyset$ **then** return **false** ;
12 **else** $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C \wedge c' \neq c \wedge x_i, x_j \in X(c') \wedge j \neq i\}$;
13 return **true** ;
end

algorithms such as AC1 or AC2), the algorithm maintains a list Q of all the pairs (x_i, c) for which we are not guaranteed that $D(x_i)$ is arc consistent on c. In line 7, Q is filled with all possible pairs (x_i, c) such that $x_i \in X(c)$. Then, the main loop (line 8) picks the pairs (x_i, c) in Q one by one (line 9) and calls `Revise`(x_i, c) (line 10). If $D(x_i)$ is wiped out, the algorithm returns false (line 11). Otherwise, if $D(x_i)$ is modified, it can be the case that a value for another variable x_j has lost its support on a constraint c' involving both x_i and x_j. Hence, all pairs (x_j, c') such that $x_i, x_j \in X(c')$ must be put again in Q (line 12). When Q is empty, the algorithm returns true (line 13) as we are guaranteed that all arcs have been revised and all remaining values of all variables are consistent with all constraints

Proposition 27 (GAC3). *GAC3 is a sound and complete algorithm for achieving arc consistency that runs in $O(er^3d^{r+1})$ time and $O(er)$ space, where r is the greatest arity among constraints.*

McGregor proposed a different way of propagating constraints in AC3, that was later named *variable-oriented*, as opposed to the arc-oriented propagation policy of AC3 [91]. Instead of putting in Q all arcs that should be revised after a change in $D(x_i)$ (line 12), we simply put x_i. Q contains variables for which a change in their domain has not yet been propagated. When picking a variable x_j from Q, the algorithm revises all arcs (x_i, c) that could lead to further deletions because of x_j. The implementation of this version of AC3 is simpler because

the elements in Q are just variables. But this less precise information has a drawback. An arc can be revised several times whereas the classical AC3 would revise it once. For instance, imagine a network containing a constraint c with scheme (x_1, x_2, x_3). If a modification occurs on x_2 because of a constraint c', AC3 puts (x_1, c) and (x_3, c) in Q. If a modification occurs on x_3 because of another constraint c'' while the previous arcs have not yet been revised, AC3 adds (x_2, c) to Q but not (x_1, c) which is already there. The same scenario with McGregor's version will put x_2 and x_3 in Q. Picking them from Q in sequence, it will revise (x_1, c) and (x_3, c) because of x_2, and (x_1, c) and (x_2, c) because of x_3. (x_1, c) has been revised twice. Boussemart et al. proposed a modified version of McGregor's algorithm that solves this problem by storing a counter for each arc [28].

From now on, I switch to binary normalized networks because most of the literature used this simplification, and I do not want to make assumptions on which extension the authors would have chosen. Nevertheless, the ideas always allow extensions to non normalized binary networks, and most of the time to networks with non-binary constraints.

Corollary 28 (AC3). *AC3 achieves arc consistency on binary networks in $O(ed^3)$ time and $O(e)$ space.*

The time complexity of AC3 is not optimal. The fact that function `Revise` does not remember anything about its computations to find supports for values leads AC3 to do and redo many times the same constraint checks.

Example 29. Let x, y and z be three variables linked by the constraints $c_1 \equiv x \leq y$ and $c_2 \equiv y \neq z$, with $D(x) = D(y) = \{1, 2, 3, 4\}$ and $D(z) = \{3\}$. `Revise`(x, c_1) requires 1 constraint check for finding support for $(x, 1)$, 2 checks for $(x, 2)$, etc., so a total of 1+2+3+4=**10** constraint checks to prove that all values in $D(x)$ are consistent with c_1. All these constraint checks are depicted as arrows in Fig. 3.a. `Revise`(y, c_1) requires **4** additional constraint checks to prove that y values are all consistent with $(x, 1)$. `Revise`(y, c_2) requires **4** constraint checks to prove that all values are consistent with $(z, 3)$ except $(y, 3)$ which is removed. Hence, the arc (x, c_1) is put in Q. `Revise`(z, c_2) requires **1** single constraint check to prove that $(z, 3)$ is consistent with $(y, 1)$.

When (x, c_1) is picked from Q, a new call to `Revise`(x, c_1) is launched (Fig. 3.b). It requires 1+2+3+3=**9** checks, among which only $((x, 3), (y, 4))$ has not already been performed at the first call.

4.2.2 AC4

AC3 being non optimal, Mohr and Henderson proposed AC4 to improve the time complexity [92, 93]. The idea of AC4, as opposed to AC3, is to store a lot of information. AC3 performs the minimum amount of work inside a call to `Revise`, just ensuring that all remaining values of x_i are consistent with c and memorizing nothing. The price to pay is to redo much of the work if the same `Revise` is recalled. AC4 stores the maximum amount of information in a

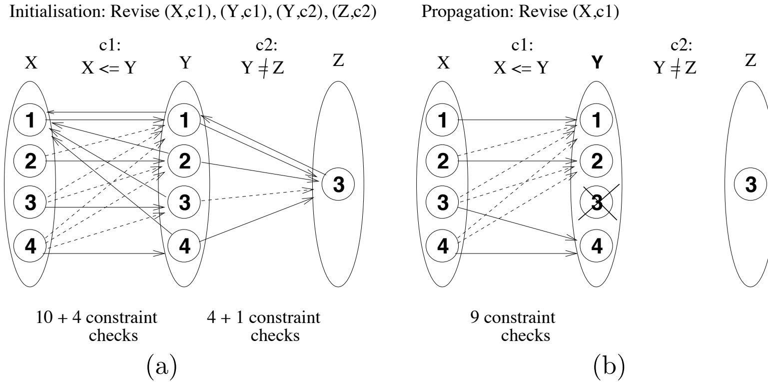

Figure 3: AC3 behavior depicted on Example 29. (Plain arrows represent positive constraint checks whereas dashed arrows represent negative ones.)

preprocessing step in order to avoid redoing several times the same constraint check during the propagation of deletions.

AC4 is presented in Algorithm 2. It computes a counter $\texttt{counter}[x_i, v_i, x_j]$ for each triple (x_i, v_i, x_j) where $c_{ij} \in C$ and $v_i \in D(x_i)$. This counter will finally say how many supports v_i has on c_{ij}. AC4 also builds lists $S[x_j, v_j]$ containing all values that are supported by (x_j, v_j) on c_{ij}. In the initialization phase, AC4 performs all possible constraint checks on all constraints. Each time a support $v_j \in D(x_j)$ is found for (x_i, v_i) on c_{ij}, $\texttt{counter}[x_i, v_i, x_j]$ is incremented, and (x_i, v_i) is added to $S[x_j, v_j]$ (lines 3 and 5). Each time a value is found without support on a constraint, it is removed from the domain and put in the list Q for future propagation (line 4). Once the initialization is finished, we enter the propagation loop (line 7), which consists in propagating the consequences of the removals of values in Q. For each value (x_j, v_j) picked from Q (line 8), we just need to decrement $\texttt{counter}[x_i, v_i, x_j]$ for each value $(x_i, v_i) \in S[x_j, v_j]$ to maintain the counters up to date (line 11). If $\texttt{counter}[x_i, v_i, x_j]$ reaches zero, this means that (x_j, v_j) was the last support for (x_i, v_i) on c_{ij}. (x_i, v_i) is removed and put in the list Q (lines 12 and 13). When Q is empty, we know that all values remaining in the domains have a non zero counter on all their constraints, and so are arc consistent.

AC4 is the first algorithm in a category later named 'fine-grained' algorithms [127] because they perform propagations (via list Q) at the level of values. 'Coarse-grained' algorithms, such as AC3, propagate at the level of constraints (or arcs), which is less precise and can involve unnecessary work.

Proposition 30 (AC4). *AC4 achieves arc consistency on binary normalized networks in $O(ed^2)$ time and $O(ed^2)$ space. Its time complexity is optimal.*

Example 31. Take again the network in Example 29 with constraints $c_1 \equiv x \leq y$ and $c_2 \equiv y \neq z$, and domains $D(x) = D(y) = \{1, 2, 3, 4\}$ and $D(z) = \{3\}$. In its initialization phase, AC4 first counts the number of supports of each

Algorithm 2: AC4

```
function AC4(in X: set): Boolean ;
   begin
       /* initialization */;
 1     Q ← ∅; S[x_j, v_j] = 0, ∀v_j ∈ D(x_j), ∀x_j ∈ X;
 2     foreach x_i ∈ X, c_ij ∈ C, v_i ∈ D(x_i) do
 3         initialize counter[x_i, v_i, x_j] to |{v_j ∈ D(x_j) | (v_i, v_j) ∈ c_ij}|;
 4         if counter[x_i, v_i, x_j] = 0 then remove v_i from D(x_i) and add (x_i, v_i) to
           Q;
 5         add (x_i, v_i) to each S[x_j, v_j] s.t. (v_i, v_j) ∈ c_ij;
 6         if D(x_i) = ∅ then return false ;
       /* propagation */;
 7     while Q ≠ ∅ do
 8         select and remove (x_j, v_j) from Q;
 9         foreach (x_i, v_i) ∈ S[x_j, v_j] do
10             if v_i ∈ D(x_i) then
11                 counter[x_i, v_i, x_j] = counter[x_i, v_i, x_j] − 1;
12                 if counter[x_i, v_i, x_j] = 0 then
13                     remove v_i from D(x_i); add (x_i, v_i) to Q;
14                     if D(x_i) = ∅ then return false ;
15     return true ;
   end
```

value on each constraint and builds the lists of supported values. Thus, in its initialization, AC4 performs all possible constraint checks for every value in each domain, that is, $4 \cdot 4 =$ **16** constraint checks on c_1 and $4 \cdot 1 =$ **4** on c_2.[5] At the end of this phase, the data structures are the following:

$$\begin{array}{lll}
\texttt{counter}[x,1,y]=4 & \texttt{counter}[y,1,x]=1 & \texttt{counter}[y,1,z]=1 \\
\texttt{counter}[x,2,y]=3 & \texttt{counter}[y,2,x]=2 & \texttt{counter}[y,2,z]=1 \\
\texttt{counter}[x,3,y]=2 & \texttt{counter}[y,3,x]=3 & \texttt{counter}[y,3,z]=0 \\
\texttt{counter}[x,4,y]=1 & \texttt{counter}[y,4,x]=4 & \texttt{counter}[y,4,z]=1 \\
 & & \texttt{counter}[z,3,y]=3
\end{array}$$

$$\begin{array}{ll}
S[x,1]=\{(y,1),(y,2),(y,3),(y,4)\} & S[y,1]=\{(x,1),(z,3)\} \\
S[x,2]=\{(y,2),(y,3),(y,4)\} & S[y,2]=\{(x,1),(x,2),(z,3)\} \\
S[x,3]=\{(y,3),(y,4)\} & S[y,3]=\{(x,1),(x,2),(x,3)\} \\
S[x,4]=\{(y,4)\} & S[y,4]=\{(x,1),(x,2),(x,3),(x,4),(z,3)\} \\
 & S[z,3]=\{(y,1),(y,2),(y,4)\}
\end{array}$$

The only counter equal to zero is $\texttt{counter}[y,3,z]$. So, $(y,3)$ is removed and AC4 enters the propagation loop with $(y,3)$ in Q. When $(y,3)$ is picked from Q, $S[y,3]$ is traversed and $\texttt{counter}[x,1,y]$, $\texttt{counter}[x,2,y]$,$\texttt{counter}[x,3,y]$ are

[5] In the original version of AC4 presented in [92], each constraint c_{ij} is processed twice (once for x_i and once for x_j), which gives 32 constraint checks on c_1 and 8 on c_2). The general version presented in [93] processes each constraint only once, updating all relevant counters and lists at the same time.

decremented (because $(x,1),(x,2),(x,3)$ are in $S[y,3]$). None of these counters are equal to zero and no value is removed. We observe that the propagation of the deletion of $(y,3)$ did not require any constraint check. It required traversals of $S[..]$ lists and updates of counters.

While being optimal in time, AC4 does not only suffer from its high space complexity. Its very expensive initialization phase can be by itself prohibitive in time. In fact, we can informally say that AC4 has optimal worst-case time complexity but it almost always reaches this worst-case. Wallace discussed this issue in [121]. In addition, even when the initialization phase has finished, AC4 maintains a so accurate view of the process that it spends a lot of effort updating its counters and traversing its lists. This is visible in Example 31, where the removal of $(y,3)$ provoked traversal of $S[y,3]$ and counter updates, whereas all remaining values had supports.

The non-binary version GAC4, proposed by Mohr and Masini in [93], is in the optimal $O(erd^r)$ time complexity given in Section 4.1, where r is the greatest arity among all constraints.

4.2.3 AC6

Bessiere and Cordier proposed AC6, a compromise between AC3 laziness and AC4 eagerness [15, 14]. The motivation behind AC6 is both to keep the optimal worst-case time complexity of AC4 and to stop the search for support for a value on a constraint as soon as the first support is found, as done in `Revise` of AC3. In addition, AC6 maintains a data structure lighter than AC4. In fact, the idea in AC6 is not to count all the supports a value has on a constraint, but just to ensure that it has *at least* one. AC6 only needs lists S, where $S[x_j,v_j]$ contains all values for which (x_j,v_j) is the *current* support. That is, $(x_i,v_i) \in S[x_j,v_j]$ if and only if v_j was the first support found for v_i on c_{ij}. [6]

In Algorithm 3, AC6 looks for one support (the *first* one or *smallest* one with respect to the ordering on integers) for each value (x_i,v_i) on each constraint c_{ij} (line 3). When (x_j,v_j) is found as the smallest support of (x_i,v_i) on c_{ij}, (x_i,v_i) is added to $S[x_j,v_j]$, the list of values currently having (x_j,v_j) as smallest support (line 4). If no support is found, (x_i,v_i) is removed and is put in the list Q for future propagation (line 5). The propagation loop (line 7) consists in propagating the consequences of the removal of values in Q. When (x_j,v_j) is picked from Q, AC6 looks for the *next* support on c_{ij} for each value (x_i,v_i) in $S[x_j,v_j]$. Instead of starting at $min_D(x_j)$ as AC3 would do, it starts at the value of $D(x_j)$ following v_j (line 11). If a new support v'_j is found, (x_i,v_i) is put in $S[x_j,v'_j]$ (line 12). Otherwise, (x_i,v_i) is removed and put in Q (line 14). When Q is empty, we know that all remaining values have a current support on every constraint.

Like AC4, AC6 is a fine-grained algorithm because it propagates along values. It does not reconsider constraint c_{ij} when the removed value (x_j,v_j) has no

[6] A similar technique, called 'watch literals', has independently been proposed by Moskewicz et al. for efficient unit propagation in their Chaff solver for SAT [97].

Algorithm 3: AC6

```
function AC6(in X: set): Boolean ;
   begin
       /* initialization */;
 1     Q ← ∅; S[x_j, v_j] = 0, ∀v_j ∈ D(x_j), ∀x_j ∈ X;
 2     foreach x_i ∈ X, c_ij ∈ C, v_i ∈ D(x_i) do
 3         v_j ← smallest value in D(x_j) s.t. (v_i, v_j) ∈ c_ij;
 4         if v_j exists then add (x_i, v_i) to S[x_j, v_j];
 5         else remove v_i from D(x_i) and add (x_i, v_i) to Q;
 6         if D(x_i) = ∅ then return false ;
       /* propagation */;
 7     while Q ≠ ∅ do
 8         select and remove (x_j, v_j) from Q;
 9         foreach (x_i, v_i) ∈ S[x_j, v_j] do
10             if v_i ∈ D(x_i) then
11                 v'_j ← smallest value in D(x_j) greater than v_j s.t. (v_i, v_j) ∈ c_ij;
12                 if v'_j exists then add (x_i, v_i) to S[x_j, v'_j];
13                 else
14                     remove v_i from D(x_i); add (x_i, v_i) to Q;
15                     if D(x_i) = ∅ then return false ;
16     return true ;
   end
```

chance to provoke another removal in $D(x_i)$, that is, when $D(x_i) \cap S[x_j, v_j] = \emptyset$.

Proposition 32 (AC6). *AC6 achieves arc consistency on binary normalized networks in $O(ed^2)$ time and $O(ed)$ space.*

Example 33. I show what the data structures of AC6 are on the example used for AC3 (Example 29) and for AC4 (Example 31), i.e., constraints $c_1 \equiv x \leq y$ and $c_2 \equiv y \neq z$ and domains $D(x) = D(y) = \{1, 2, 3, 4\}$ and $D(z) = \{3\}$. In its initialization phase, AC6 looks for *one* support (the smallest) for each value on each constraint and stores the fact that a value (x_j, v_j) has been found as supporting (x_i, v_i) by adding (x_i, v_i) to $S[x_j, v_j]$. Thus, in its initialization, AC6 performs the same number of constraint checks as AC3, namely 10+4 on c_1 and 4+1 on c_2. At the end of this phase, the data structures are the following,

$$S[x,1] = \{(y,1),(y,2),(y,3),(y,4)\} \qquad S[y,1] = \{(x,1),(z,3)\}$$
$$S[x,2] = \{\} \qquad S[y,2] = \{(x,2)\}$$
$$S[x,3] = \{\} \qquad S[y,3] = \{(x,3)\}$$
$$S[x,4] = \{\} \qquad S[y,4] = \{(x,4)\}$$
$$S[z,3] = \{(y,1),(y,2),(y,4)\}$$

and the list Q contains $(y, 3)$ which has been removed. When AC6 enters the propagation loop it pops $(y, 3)$ from Q, $S[y, 3]$ is traversed and a new support *greater than* 3 is sought for $(x, 3)$. $(3, 4) \in c_1(x, y)$, so $(x, 3)$ is added to $S[y, 4]$, which supports now both $(x, 3)$ and $(x, 4)$. The deletion of $(y, 3)$ required a single

Algorithm 4: Function `Revise` for AC2001

```
function Revise2001(in x_i: variable; c_ij: constraint): Boolean ;
  begin
1     CHANGE ← false;
2     foreach v_i ∈ D(x_i) s.t. Last(x_i, v_i, x_j) ∉ D(x_j) do
3         v_j ← smallest value in D(x_j) greater than Last(x_i, v_i, x_j) s.t.
          (v_i, v_j) ∈ c_ij;
4         if v_j exists then Last(x_i, v_i, x_j) ← v_j;
5         else
6             remove v_i from D(x_i);
7             CHANGE ← true;
8     return CHANGE ;
  end
```

constraint check and the traversal of list $S[y, 3]$. Note that $S[y, 3]$ contained less values than in AC4 because AC6 stores a single support per value.

4.2.4 AC2001

In fine-grained algorithms, such as AC4 or AC6, the propagation is value-oriented. The deletion of a value (x_j, v_j) is directly propagated through Q on values (x_i, v_i) that had (x_j, v_j) as support (that is, on values (x_i, v_i) that are in $S[x_j, v_j]$). Coarse-grained algorithms are arc-oriented. They do not propagate the consequences of value removals to other values. They propagate changes in the domain of a variable x_j on the other variables x_i sharing a constraint c with x_j: List Q contains pairs (x_i, c) for which some variable x_j in $X(c)$ has changed. Although coarse-grained algorithms are less precise in the way they propagate, they have a double advantage. First, the architecture of constraint solvers (see Section 9) usually supports an arc-oriented propagation and not a value-oriented one. Second, all fine-grained algorithms require lists $S[..]$ of supported values as data structure, which is more complex to implement and maintain. These were the motivations for AC2001, the first (and only) optimal coarse-grained algorithm [24, 127, 25].

AC2001 follows the same framework as AC3, but achieves optimality by storing the smallest support for each value on each constraint, like AC6. However, the way this information is stored and used differs from that in AC6. AC2001 does not use lists $S[x_j, v_j]$ to store those (x_i, v_i) that have v_j as smallest support on c_{ij}. It uses a pointer `Last`$[x_i, v_i, x_j]$ that contains v_j.

AC2001 differs from AC3 only by its `Revise` function and by its initialization phase which needs to initialize the pointers `Last`$[x_i, v_i, x_j]$ to some dummy value smaller than $min_D(x_j)$. In `Revise2001` (Algorithm 4), when a value v_j in $D(x_j)$ is found to support (x_i, v_i) on c_{ij}, AC2001 assigns v_j to `Last`$[x_i, v_i, x_j]$ (line 4). The next time (x_i, c_{ij}) will be revised, supports will be sought for (x_i, v_i) only if `Last`$[x_i, v_i, x_j]$ is no longer in $D(x_j)$ (line 2). More importantly, optimality is obtained because values in $D(x_j)$ that are smaller than `Last`$[x_i, v_i, x_j]$ are

not checked again because they were already unsuccessfully checked in previous calls to `Revise2001` (line 3).

Proposition 34 (AC2001). *AC2001 achieves arc consistency on binary normalized networks in $O(ed^2)$ time and $O(ed)$ space.*

Example 35. Again I show the data structures of AC2001 on the example used for the other algorithms, i.e., constraints $c_1 \equiv x \leq y$ and $c_2 \equiv y \neq z$ and domains $D(x) = D(y) = \{1, 2, 3, 4\}$ and $D(z) = \{3\}$. In its initialization phase, AC2001 looks for the smallest support for each value on each constraint and stores it in the `Last` structure. It performs exactly the same constraint checks as AC3 or AC6. At the end of this phase, the data structures are the following,

$$
\begin{array}{lll}
\texttt{Last}[x,1,y] = 1 & \texttt{Last}[y,1,x] = 1 & \texttt{Last}[y,1,z] = 3 \\
\texttt{Last}[x,2,y] = 2 & \texttt{Last}[y,2,x] = 1 & \texttt{Last}[y,2,z] = 3 \\
\texttt{Last}[x,3,y] = 3 & \texttt{Last}[y,3,x] = 1 & \texttt{Last}[y,3,z] = nil \\
\texttt{Last}[x,4,y] = 4 & \texttt{Last}[y,4,x] = 1 & \texttt{Last}[y,4,z] = 3 \\
 & & \texttt{Last}[z,3,y] = 1
\end{array}
$$

and the list Q contains (x, c_1) because $(y, 3)$ has been removed while revising c_2. When AC2001 enters the propagation loop it pops (x, c_1) from Q, and calls `Revise`(x, c_1). It checks whether `Last`$[x, 1, y]$, `Last`$[x, 2, y]$, `Last`$[x, 3, y]$ and `Last`$[x, 4, y]$ are still in $D(y)$. `Last`$[x, 3, y]$ is no longer in $D(y)$, so a new support *greater than* 3 is sought for $(x, 3)$. $(3, 4)$ satisfies $c_1(x, y)$, so `Last`$[x, 3, y]$ receives value 4. The deletion of $(y, 3)$ required checking if the `Last` pointers of values in $D(x)$ were still in $D(y)$, and a single constraint check to find a new support for $(x, 3)$.

AC2001 can easily be extended to a GAC2001 non-binary version [25].

4.3 Other improvements

I have presented the main techniques to enforce arc consistency on a network. Other kinds of techniques exist to reduce the cost of arc consistency. They are usually added to one of the arc consistency algorithms presented above to improve its performance. I cannot be exhaustive, but here are two of those types of techniques.

4.3.1 Bidirectionality

Constraints are said to be *multidirectional* because when a tuple τ is found to support (x_i, v_i) on a constraint c, it is also a support for any $(x_j, v_j) \in \tau$ on the same constraint. The binary version of multidirectionality is called *bidirectionality*. This property, which can seem obvious, is not used as much as it could be by the algorithms presented so far.

In fact, AC3 partially uses it when it avoids putting (x_j, c) in Q after modifying x_i in `Revise`(x_i, c) (line 12 in Algorithm 1): A value v_i removed from

$D(x_i)$ had no support on c, so its removal cannot discard a support for a value in $D(x_j)$.

Gaschnig proposed to use bidirectionality more explicitly. The algorithm DEE [62] is an extension of AC3 that uses a 'Revise-both' procedure to process `Revise`(x_i, c_{ij}) and `Revise`(x_j, c_{ij}) in sequence. As a first step, Revise-both performs the same work as `Revise`(x_i, c_{ij}), but in addition, marks every value in $D(x_j)$ which has been found in a support for a value in $D(x_i)$. Once all values of x_i are checked, Revise-both revises x_j on c_{ij} by only looking for support for unmarked values of $D(x_j)$. Values marked during the first phase are guaranteed to have support. DEE does not store these marks from a call to Revise-both to another. Besides, in the propagation phase, arcs are often revised in only one direction at a time, which reduces the gain of DEE.

Van Dongen proposed a heuristic approach of using bidirectionality [115]. The algorithm AC$_b$ uses the same idea as DEE, trying to avoid work when both arcs (x_i, c_{ij}) and (x_j, c_{ij}) are in Q. AC$_b$ does not check supports in lexicographic ordering but tries to maximize the number of 'double-support' checks. A double-support check is a constraint check $c_{ij}(v_i, v_j)$ for which neither v_i nor v_j are known to be supported on c_{ij}. The motivation is that if $c_{ij}(v_i, v_j)$ is true, we deduce support for two values at the price of a single constraint check.

Bidirectionality was used even more extensively in AC7 [18, 19], an extension of AC6. Thanks to the lists of supported values of AC6, and additional pointers, AC7 fully exploits bidirectionality. This means that a constraint check $c_{ij}(v_i, v_j)$ is performed when looking for support for (x_i, v_i) on c_{ij} only if $c_{ji}(v_j, v_i)$ has never been checked while looking for supports for (x_j, v_j) on c_{ji} and there does not exist $v'_j \in D(x_j)$ such that $c_{ji}(v'_j, v_i)$ has already been successfully checked as support for (x_j, v'_j). The non-binary version of AC7 [23] is used in IlogSolver [71] to propagate general constraints. As for GAC4, it runs in the optimal $O(erd^r)$ time complexity.

Lecoutre et al. proposed several extensions of AC2001 that permit to adapt the techniques used in AC7 to coarse-grained algorithms [81]. AC3.2 is an algorithm that partially exploits bidirectionality on positive constraint checks. AC3.3 fully exploits bidirectionality on positive constraint checks. AC3.2* and AC3.3* are extensions of AC3.2 and AC3.3 that also exploit bidirectionality on negative constraint checks, like in AC7. An extensive experimentation suggests that AC3.3 is the best stand alone arc consistency algorithm, whereas AC3.2 is the best when maintained during search.

4.3.2 Ordering the propagation list

Another way of improving the time needed to enforce arc consistency is by revising first the arcs that will prune the most or that will be the cheapest to revise. In their seminal paper on the subject, Wallace and Freuder proposed several heuristics to reorder the propagation list in AC3 [122]. Among the different heuristics they analyzed, the best seemed to be the one selecting first the arcs (x_i, c_{ij}) such that the variable x_j against which to revise has the smallest domain.

Gent et al. applied to arc consistency the general criterion of 'constrainedness' defined in [65]. They proposed to select first the arc that minimizes the constrainedness κ_{ac} of arc consistency [64]. They show that this heuristic is a good way to reduce the number of constraint checks but is heavy to compute. Interestingly, approximations of their criterion give some of the good heuristics proposed by Wallace and Freuder.

The most comprehensive study on ordering heuristics for coarse-grained arc consistency algorithms was recently proposed by Boussemart et al. in [28]. They not only studied heuristics to reorder the propagation list Q, but also the type of information we put in it. Q can be a list of arcs to revise, as in regular AC3 (arc-oriented revision), a list of variables whose domain has been modified as in McGregor's version (variable-oriented revision), or a list of constraints which had a variable of their scheme modified. Lists of variables being much shorter than lists of arcs, they showed that heuristics handling Q are less time consuming when incorporated in variable-oriented implementations. Since McGregor's algorithm suffers from redundant revisions (see Subsection 4.2.1), Boussemart et al. proposed a modified version that avoids these redundant revisions while keeping the advantage of variable-oriented revision. As for saving constraint checks, they found that several heuristics close to that already proposed by Wallace and Freuder or by van Dongen [122, 114] show good performance. Among all, they recommend a variable-oriented implementation of coarse-grained algorithms (they experimented with AC3.2) in which the variable with the smallest domain is picked first from Q.

5 Higher Order Consistencies

In Section 4, we have seen that arc consistency, which is the most natural technique for tightening a network, has received great attention from the community. Nevertheless, this is not the only way to tighten a network, and as early as in the 70's, several authors proposed techniques that discover more inconsistencies than arc consistency.

5.1 Path consistency

Path consistency was proposed by Montanari as a necessary condition for the consistency of pairs of values in binary normalized networks [95]. Roughly speaking, it says that if for a given pair of values (v_i, v_j) on a pair of variables (x_i, x_j) there exists a sequence of variables from x_i to x_j such that we cannot find a sequence of values for these variables starting at v_i and finishing at v_j, and satisfying all binary constraints along the sequence, then (v_i, v_j) is inconsistent.

Definition 36 (Path consistency). *Let $N = (X, D, C)$ be a normalized network.*

- *Given two variables x_i and x_j in X, the pair of values $(v_i, v_j) \in D(x_i) \times D(x_j)$ is path consistent iff for any sequence of variables $Y = (x_i =$*

$x_{k_1}, x_{k_2}, \ldots, x_{k_p} = x_j$) such that for all $q \in [1..p-1], c_{k_q,k_{q+1}} \in C$, there exists a tuple of values $(v_i = v_{k_1}, v_{k_2}, \ldots, v_{k_p} = v_j) \in \pi_Y(D)$ such that for all $q \in [1..p-1], (v_{k_q}, v_{k_{q+1}}) \in c_{k_q,k_{q+1}}$.

- *The network N is* path consistent (PC) *iff for any pair of variables $(x_i, x_j), i \neq j$, any locally consistent pair of values on (x_i, x_j) is path consistent.*

Example 37. Consider the network N with variables x_1, x_2, x_3, domains $D(x_1) = D(x_2) = D(x_3) = \{1, 2\}$, and $C = \{x_1 \neq x_2, x_2 \neq x_3\}$. N is not path consistent because neither $((x_1, 1), (x_3, 2))$ nor $((x_1, 2), (x_3, 1))$ can be extended to a value of x_2 satisfying both c_{12} and c_{23}. The network $N' = (X, D, C \cup \{x_1 = x_3\})$ is path consistent.

Montanari observed that it is sufficient to enforce path consistency only on paths of length 2 to obtain the same level of local consistency as path consistency.

Definition 38 (2-path consistency). *Let $N = (X, D, C)$ be a normalized network.*

- *Given two variables x_i and x_j in X, the pair of values $(v_i, v_j) \in D(x_i) \times D(x_j)$ is 2-path consistent iff for any third variable $x_k \in X$ with $c_{ik} \in C$ and $c_{kj} \in C$, there exists a value $v_k \in D(x_k)$ such that $(v_i, v_k) \in c_{ik}$ and $(v_j, v_k) \in c_{kj}$.*
- *The network N is* 2-path consistent *iff for any pair of variables $(x_i, x_j), i \neq j$, any locally consistent pair of values on (x_i, x_j) is 2-path consistent.*

Proposition 39. *Path consistency and 2-path consistency are equivalent.*

Path consistency does not reduce domains of variables but removes pairs of values. As a result, the path consistent closure of a normalized network N is not in $\mathcal{P}_{ND}^{sol}$. I define $\mathcal{P}_{N2}$ as the subset of $\mathcal{P}_N$ where networks are normalized and differ from N only by adding or tightening binary constraints. The path consistent closure $PC(N)$ of N is the union (according to $\sqcup_N$) of all path consistent networks in $\mathcal{P}_{N2}$. In $\mathcal{P}_{N2}$, there can be several networks nogood-equivalent to $PC(N)$ because constraints with the same scheme can differ on non valid tuples, which does not change the set of locally inconsistent instantiations. Nevertheless, PC algorithms represent modified constraints extensionally, generating only embedded constraints. So, if we consider networks where all binary constraints are embedded, the relation $\preceq$ is a partial order on $\mathcal{P}_{N2}$ and PC algorithms are guaranteed to converge on $PC(N)$.

Several algorithms achieving PC were proposed in the literature. Each time a new technique was proposed for arc consistency, it was soon applied to path consistency. PC1 [95, 87] can be seen as the path consistency counterpart of the brute-force AC1. PC2 is the extension of AC3 to path consistency [87]. PC3 [92] and PC4 [68] use lists of support, like AC4, to reach optimality. PC5 [112] and PC6 [32] extend AC6. PC7 [31] and PC8 [33] are simplifications of PC6 that perform well in practice. PC5++ [112] applies bidirectionality of AC7. Finally, PC2001 [127, 25] extends AC2001.

A drawback of path consistency is that enforcing it can produce additional constraints that were not in C_N (see Example 37). Furthermore, even when a constraint $c(x_i, x_j)$ is already in C_N, its refinement by PC can impose to change its semantics and to represent this new constraint extensionally whereas it was given as a function.

Example 40. Consider the network with variables x_1, x_2, x_3, domains $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3, 4\}$, and $C = \{|x_1 - x_2| \geq 2, x_2 \neq x_3, x_1 \neq x_3\}$. These three constraints can be given by their arithmetic expression if the constraint toolkit in use permits them. However, enforcing PC will discard the tuples $(2, 4)$, and $(3, 1)$ from c_{13}, which probably requires a storage in extension of this constraint. If c_{13} had a specific propagation algorithm for enforcing AC on it (see Section 9), it no longer works on this new constraint.

The last thing we can notice is that even if path consistency is usually considered in binary normalized networks, nothing in Definition 36 prevents its use on non-binary normalized networks. Non-binary constraints are just ignored.

5.2 k-consistencies

A few years after Montanari's paper, Freuder extended the notion of local consistencies stronger than AC to a whole class of consistencies, called k-consistencies [53, 54].

Definition 41 (k-consistency). *Let $N = (X, D, C)$ be a network.*

- *Given a set of variables $Y \subseteq X$ with $|Y| = k - 1$, a locally consistent instantiation I on Y is k-consistent iff for any kth variable $x_{i_k} \in X \setminus Y$ there exists a value $v_{i_k} \in D(x_{i_k})$ such that $I \cup \{(x_{i_k}, v_{i_k})\}$ is locally consistent.*
- *The network N is* k-consistent *iff for any set Y of $k - 1$ variables, any locally consistent instantiation on Y is k-consistent.*

Given a normalized network N, $\mathcal{P}_{Nk}$ denotes the subset of $\mathcal{P}_N$ containing all normalized networks N' in which only constraints of arity k can differ from N. More formally, $N' \in \mathcal{P}_{Nk}$ if and only if $N' \in \mathcal{P}_N$, $D_{N'} = D_N$, and any constraint in $C_{N'} \setminus C_N$ has arity k. The k-consistent closure of N is the union (according to $\sqcup_N$) of all k-consistent networks in $\mathcal{P}_{N(k-1)}$. (Enforcing k-consistency makes explicit nogoods of size $k - 1$.) I restrict to normalized networks because $\sqcup_N$ is not defined on arbitrary networks (see Section 3). $\preceq$ is a partial order on $\mathcal{P}_{N(k-1)}$ only if all constraints of arity $k - 1$ are embedded.

As observed by Dechter [46], even if 3-consistency has strong similarities with (2-)path consistency, it is not equivalent. Indeed, 3-consistency ensures that any instantiation of length 2 can be extended to an instantiation involving any third variable without violating *any* constraint, whereas (2-)path consistency only guarantees that *binary* constraints are not violated.

Example 42. Suppose a network involving variables x_1, x_2, x_3 with domains $D(x_1) = D(x_1) = D(x_1) = \{1, 2\}$, and a single constraint $c(x_1, x_2, x_3) = \{(1, 1, 1), (2, 2, 2)\}$. This network is path consistent because it does not contain any binary constraint. It is *not* 3-consistent because the instantiation $(x_1 = 1, x_2 = 2)$, which is locally consistent, cannot be extended consistently to x_3. 3-consistency produces the three binary constraints $c_{12} = \{(1, 1), (2, 2)\}$, $c_{23} = \{(1, 1), (2, 2)\}$ and $c_{13} = \{(1, 1), (2, 2)\}$.

k-consistency ensures that each time we have a locally consistent instantiation of size $k-1$, we can consistently extend it to any kth variable. So, the question is 'how to build locally consistent instantiations of size $k-1$?'. Strong k-consistencies are properties that guarantee that the network is j-consistent for $1 \leq j \leq k$. Thus, we can build from scratch a locally consistent instantiation of size k without any backtrack.

Definition 43 (Strong k-consistency). *A network is* strongly k-consistent *iff it is j-consistent for all $j \leq k$.*

Given a normalized network N, $\mathcal{P}^*_{Nk}$ denotes the subset of $\mathcal{P}_N$ containing all normalized networks N' in which only the domains and constraints of arity at most k can differ from N. More formally, $N' \in \mathcal{P}^*_{Nk}$ if and only if $N' \in \mathcal{P}_N$, $D_{N'} \subseteq D_N$, and any constraint in $C_{N'} \setminus C_N$ has arity at most k. The strong k-consistent closure of N is the union of all strongly k-consistent networks in $\mathcal{P}^*_{N(k-1)}$. The relation $\preceq$ is not a partial order in $\mathcal{P}^*_{N(k-1)}$ even if we restrict to embedded constraints. As a consequence, an algorithm achieving strong k-consistency by iteratively enforcing j-consistency, $1 \leq j \leq k$, is not guaranteed to terminate on the strong k-consistent closure of N. It may terminate on a network of $\mathcal{P}^*_{N(k-1)}$ nogood-equivalent to the closure.

Example 44. Consider the network with variables $x_1, \ldots, x_6$, domains equal to $\{1, 2\}$ and $C = \{c_1(x_1, x_2, x_3, x_4),\ c_2(x_2, x_3, x_4, x_5),\ x_2 = x_6,\ x_6 \neq x_3\}$, with $c_1 = \{(1112), (1121), (1211), (2122), (2212), (2221)\}$ and $c_2 = \{(1112), (1211), (1222), (2112), (2122), (2222)\}$. If we apply 4-consistency on x_2, x_3, x_4 wrt x_1 and x_5, we derive the constraint $c_3(x_2, x_3, x_4) = \{(121), (122), (211), (212)\}$. 3-consistency on x_2, x_3 wrt x_4 produces the constraint $c_4(x_2, x_3) = \{(12), (21)\}$. By applying first 3-consistency to x_2, x_3 wrt x_6, the constraint c_4 would have been produced before c_3. So c_3 would have never been generated because all its tuples are already inconsistent with c_4.

The algorithms proposed by Freuder and Cooper in [53, 37] both reach a fixpoint which is not the strong k-consistent closure. They make all constraints (up to arity $k-1$) as explicit as possible. For instance, if a pair of values $((x_i, v_i), (x_j, v_j))$ is path inconsistent, they create a constraint on every superset Y of $\{x_i, x_j\}$ with $|Y| < k$, and this constraint forbids all tuples τ on Y where $\tau[(x_i, x_j)] = (v_i, v_j)$. Cooper showed that his algorithm runs in $O(n^k d^k)$, which is the optimal time complexity for strong k-consistency. The algorithm proposed by Cooper requires $O(n^k d^k)$ space. The optimal space complexity for strong k-consistency is $O(n^{k-1} d^{k-1})$ because we must store all the constraints of arity

$k-1$ that k-consistency creates each time an instantiation of size $k-1$ does not extend to a kth variable. .

I said in Section 3 that the maximal amount of simplification we can perform on a network is to reach a globally consistent network, that is, a network on which *all* locally consistent instantiations can be extended to solutions. Strong n-consistency guarantees that.

Proposition 45. *If a network is strongly n-consistent then it is globally consistent.*

Enforcing global consistency on an arbitrary network is far too space consuming (in $O(n^{n-1}d^{n-1})$). Freuder gave conditions on the associated hypergraph for which strong k-consistency ($k < n$) is sufficient to allow a backtrack-free search [54]. In [47], Dechter and Pearl developed *adaptive consistency (AdC)*, a technique inspired from dynamic programming. Given a total ordering on the variables, AdC adapts the level of k-consistency enforced on each variable x_i depending on the number of variables that share a constraint with x_i and that precede it in the ordering. The obtained network guarantees backtrack-free search. (See Chapter 5 in [109].)

In [55], Freuder proposed (i, j)-consistency, a generalization of k-consistency where we do not guarantee that instantiations of size $k-1$ can be extended to instantiations of size k, but instantiations of size i can be extended to j additional variables. k-consistency is $(k-1, 1)$-consistency. Since the main drawback of k-consistencies is the huge space they require to store all forbidden instantiations of size $k-1$, we can design local consistencies requiring less space by setting i to a small value in (i, j)-consistency.

5.3 Montanari's decomposability and minimality

Montanari characterized networks that can be made globally consistent in polynomial space. These are networks for which the set of solutions is a decomposable relation [95], also named binary decomposable relation in [46].

Definition 46 (Decomposable in the sense of Montanari).

- *A relation ρ with scheme X is binary-representable iff there exists a binary network N, $X_N = X$, such that $sol(N) = \rho$.*
- *A relation ρ with scheme X is decomposable in the sense of Montanari iff for all $Y \subseteq X$, $\pi_Y(\rho)$ is binary-representable.*
- *A network N is decomposable in the sense of Montanari iff $sol(N)$ is a decomposable relation.*

Example 47. ([95]) Consider the network N in Fig. 4, with variables x_1, x_2, x_3, x_4, domains $D(x_1) = \{1, 2, 3, 4\}$, $D(x_2) = D(x_3) = D(x_4) = \{0, 1\}$, and constraints as shown in the boxes on the figure. $sol(N)$ is equal to the relation R on the top right-hand corner of the figure, which is thus representable by a binary network. However, R is not decomposable in the sense of Montanari

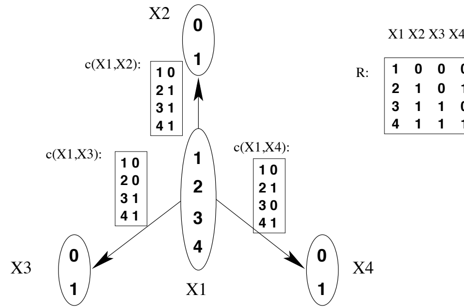

Figure 4: A relation R that can be represented by a binary network but which is non decomposable in the sense of Montanari.

because $\pi_{\{x_2,x_3,x_4\}}(R) = \{(000), (101), (110), (111)\} = \{x_2 = x_3 \vee x_4\}$ cannot be represented by a binary network. (Any binary network on x_2, x_3, x_4 that accepts all tuples in the relation also accepts the tuple (100).)

Proposition 48. *If a network $N = (X, D, C)$ is decomposable in the sense of Montanari then there exists a binary network $G_N = (X, D, C_G)$, which is globally consistent and $sol(N) = sol(G_N)$.*

Decomposability of Montanari is stronger than what is commonly called 'decomposable constraint'. (See [63] or Section 9.2 for more details.)

Example 47 shows that it is not because a network is binary that it is decomposable in the sense of Montanari. For binary networks, Montanari proposed the concept of *minimal network*, which is the best approximating binary network for global consistency. This is thus another technique for tightening binary networks.

Definition 49 (Minimal network). *Given a binary network $N = (X, D, C)$, the* minimal network *of N is the binary normalized and embedded in D network $M_N = (X, D, C_M)$ such that any locally consistent instantiation of length 2 is globally consistent and $sol(M_N) = sol(N)$.*

Corollary 50. *Given a binary network N, if $sol(N)$ is decomposable in the sense of Montanari, the minimal network M_N is globally consistent.*

Minimality on a binary network could be considered as a kind of local consistency. But local consistencies usually refer to properties which are polynomial to enforce. Building the minimal network is obviously intractable because once we have the minimal network, it is constant time to decide consistency of the original network (by checking non emptiness of any constraint).

The question of building the minimal network was called the 'central problem' by Montanari. This led to some confusion as it was sometimes believed that generating a solution is polynomial if the network is minimal. Dechter partially fixed the ambiguity by saying that:

"it is still not clear, however, whether or not generating a single solution of a minimal network is hard, even though empirical experience shows that it is normally easy. Nevertheless, we do speculate that generating a single solution from the minimal network is hard..."

We can say a little more about this.

Proposition 51 (Generating solutions of a minimal network). *Generating a solution of a minimal network M_N is not backtrack-free (unless $\Pi_2^P = \Sigma_2^P$).*

Proof. The clause entailment problem is known to be non compilable[7] unless $\Pi_2^P = \Sigma_2^P$ [29]. In [38], Cros reduced the clause entailment problem to the compilability of the problem of the global consistency of a partial instantiation in a binary network N. If building solutions in a minimal network was backtrack-free, it would be polynomial to answer whether a partial instantiation is globally consistent or not. Furthermore, a minimal network has a size in $O(n^2d^2)$, which is polynomial in the size of N. Hence, the problem of the consistency of partial instantiations would be compilable, and clause entailment as well. □

5.4 Consistencies based on constraints

All consistencies I studied until now (except arc consistency) are properties of partial instantiations of variables wrt other variables. They do not take into account the network topology, i.e., which sets of variables are linked by a constraint and which are not. This is a limitation for constraint propagation, which creates new constraints everywhere in the network. This is also a limitation on non-binary networks if we want to link the level of consistency and the hypergraph structure in backtrack-free conditions. In this section, I restrict my attention to embedded networks because all the works I present used this restriction.

Janssen et al. proposed a first local consistency based on constraints instead of variables [72]. It was applied from works on relational databases [7].

Definition 52 (Pairwise consistency). *Given an embedded network N, a pair of constraints c_1 and c_2 in C_N is pairwise consistent iff any tuple on $X(c_1)$ (resp. on $X(c_2)$) satisfying c_1 (resp. c_2) can be extended to an instantiation on $X(c_1) \cup X(c_2)$ satisfying c_2 (resp. c_1), that is, iff $\pi_{X(c_1)\cap X(c_2)}(c_1) = \pi_{X(c_1)\cap X(c_2)}(c_2)$. N is* pairwise consistent *iff any pair of constraints in C_N is pairwise consistent.*

Example 53. Consider the network with variables x_1, x_2, x_3, x_4, domains $D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{1, 2\}$ and constraints $c_1(x_1, x_2, x_3) = \{(121), (211), (222)\}$ and $c_2(x_2, x_3, x_4) = \{(111), (222)\}$. This network is generalized arc consistent. However, it is not pairwise consistent because the tuple (121) from c_1 is not compatible with any tuple in c_2.

[7] The problem of asking queries of a class Q to instances of a class P is said to be *compilable* if there exists a polynomial space transformation p' of any instance p of P (the time for the transformation should just be finite) such that any query q from Q asked on p can be answered in polynomial time by using p' [29].

Janssen et al. showed in [72] that pairwise consistency is equivalent to 2-consistency on the dual encoding of the network, where dual variables represent constraints of the original network [48].

In a database context, Gyssens proposed k-wise consistency, a direct extension of pairwise consistency where we consider k constraints at a time instead of two [67]. Jégou applied this notion to constraint networks [73].

Definition 54 (k-wise consistency). *Given an embedded network N, a set of constraints $\{c_1 \dots, c_k\}$ in C_N is k-wise consistent iff for any $c_i, i \in [1..k]$, any tuple on $X(c_i)$ satisfying c_i can be extended to an instantiation on $\bigcup_{j=1}^{k} X(c_j)$ satisfying c_j for all $j \in [1..k]$, that is, iff $c_i = \pi_{X(c_i)}(\bowtie_{j=1}^{k} c_j)$. N is* k-wise consistent *iff for all $\{c_1 \dots, c_k\}$ in C_N, $\{c_1 \dots, c_k\}$ is k-wise consistent.*

k-wise consistency is the constraint-based counterpart of k-inverse consistency (see Section 6). Enforcing k-wise consistency does not alter the associated hypergraph. It just alters existing constraints.

In [74], Jégou proposed another duality between variables and constraints. He presents hyper k-consistency. This is the constraint-based counterpart of k-consistency.

Definition 55 (Hyper k-consistency). *Let N be an embedded network. A set $\{c_1, \dots, c_{k-1}\}$ of $k-1$ constraints in C_N is hyper k-consistent relative to a kth constraint c_k iff any instantiation on $\cup_{i=1}^{k-1} X(c_i)$ satisfying $c_1, \dots, c_{k-1}$ has an extension on the variables in $X(c_k)$ that satisfies c_k, that is, iff $\pi_Y(\bowtie_{i=1}^{k-1} c_i) \subseteq \pi_Y(c_k)$, where $Y = (\cup_{i=1}^{k-1} X(c_i)) \cap X(c_k)$. N is* hyper k-consistent *iff for all $\{c_1, \dots, c_{k-1}\} \subseteq C_N$, for all $c_k \in C_N$, $\{c_1, \dots, c_{k-1}\}$ is hyper k-consistent relative to c_k.*

Pairwise consistency is both 2-wise consistency and hyper 2-consistency.

Based on definition 55, Jégou characterized some sufficient conditions for a network to be consistent. These conditions link the level of hyper k-consistency of the network to the width of its hypergraph. Nevertheless, hyper k-consistency inherits one of the drawbacks of k-consistencies because enforcing hyper k-consistency creates new constraints on sets of variables that were not linked in the original network.

Dechter and van Beek proposed a new form of local consistency which is more bound to schemes of constraints already in the network than hyper k-consistency is. They refer to those new types of consistencies as *relational* consistencies [49].

Definition 56 (Relational arc consistency). *Let N be an embedded network. A constraint c in C_N is relationally arc consistent relative to a subset of variables $Y \subseteq X(c)$ iff any locally consistent instantiation on Y has an extension to a tuple on $X(c)$ that satisfies c. c is relationally arc consistent iff it is relationally arc consistent relative to every subset Y of $X(c)$. N is* relationally arc consistent *iff every constraint in C_N is relationally arc consistent.*

An advantage of relational arc consistency is that enforcing it does not create constraints between variables not linked in the original network. However, it

creates subconstraints on subsets of the schemes of the original constraints, which can be prohibitive on large arity constraints because it can create up to $2^{|X(c)|}$ subconstraints for a constraint c.

Definition 57 (Relational m-consistency). *Let N be an embedded network. A set $\{c_1, \ldots, c_m\}$ of m constraints in C_N is relationally m-consistent relative to a subset of variables $Y \subseteq \bigcup_{i=1}^{m} X(c_i)$ iff any locally consistent instantiation on Y has an extension to $\bigcup_{i=1}^{m} X(c_i)$ that satisfies $c_1, \ldots, c_m$ simultaneously. A set $\{c_1, \ldots, c_m\}$ of m constraints in C_N is relationally m-consistent iff it is relationally m-consistent relative to every subset Y of $\bigcup_{i=1}^{m} X(c_i)$. N is* relationally m-consistent *iff every set of m constraints in C_N is relationally m-consistent.*

Relational m-consistency has the same drawbacks as hyper k-consistency because it can create new constraints on any subset of variables involved in one of m constraints. Dechter and van Beek proposed a *bounded* version of relational m-consistency that permits to tackle the space and time explosion.

Definition 58 (Relational (i,m)-consistency). *Let N be an embedded network. A set of constraints $\{c_1, \ldots, c_m\} \subseteq C_N$ is relationally (i,m)-consistent iff it is relationally m-consistent relative to every subset of variables $Y \subseteq \bigcup_{i=1}^{m} X(c_i), |Y| = i$. N is* relationally (i,m)-consistent *iff every subset of m constraints in C_N is relationally (i,m)-consistent. N is* strong *relational (i,m)-consistent iff it is relationally (j,m)-consistent for every $j \leq i$.*

Relational arc consistency corresponds to strong relational $(n,1)$-consistency and relational m-consistency corresponds to strong relational (n,m)-consistency. Generalized arc consistency is relational $(1,1)$-consistency. Relational $(1,m)$-consistencies are domain-based consistencies, and so, do not modify the set of constraints.

As in the case of strong k-consistencies, algorithms enforcing strong relational (i,m)-consistencies can converge to different networks depending on the order in which they generate new constraints.

Dechter and van Beek proposed an algorithm enforcing relational (i,m)-consistency. Its complexity is exponential in $i \cdot m$. They also proposed an algorithm for adaptive relational consistency. It is inspired from adaptive consistency ([47]) and applies the right level of relational consistency to guarantee a backtrack-free search for solutions wrt a given ordering of the variables.

Walsh performed an extensive theoretical comparison of relational consistencies with k-consistencies, k-inverse consistencies and generalized arc consistency [124].

6 Domain-based Consistencies Stronger than AC

There exist local consistencies that permit to prune more values than arc consistency while keeping the set of constraints unchanged (as opposed to what is

done by k-consistencies and consistencies based on constraints —see Section 5). The first ones I present are different kinds of reasoning we can apply on triples of variables. The others involve the whole neighborhood of a variable or check local consistency of the whole network after a single assignment of a variable.

6.1 Triangle-based local consistencies

The local consistencies defined here are limited to binary normalized networks. They all deal with 'triangles' of constraints, namely triples of variables connected two-by-two by binary constraints.

The first local consistency following this line of research is *Restricted Path Consistency (RPC)*, proposed by Berlandier [13]. The motivation for RPC is to remove more inconsistent values than arc consistency whereas avoiding the cost of path consistency. Path consistency removes all pairs of values that cannot be extended to a third variable. The idea of RPC is to try to extend only those pairs of values that if removed, would lead to arc inconsistency of a value. So, in addition to arc consistency, RPC guarantees path consistency of the pairs of values $((x_i, v_i), (x_j, v_j))$ that are the only support for (x_i, v_i) on c_{ij}. If such a pair is path inconsistent, its deletion would lead to the arc inconsistency of (x_i, v_i). Thus (x_i, v_i) can be removed. These few additional path consistency checks allow the detection of more inconsistent values than arc consistency without having to delete any pair of values, and so leaving the structure of the network unchanged.

Definition 59 (Restricted path consistency). *A binary normalized network $N = (X, D, C)$ is* restricted path consistent (RPC) *iff it is arc consistent and for all $x_i \in X$, for all $v_i \in D(x_i)$, for all $c_{ij} \in C$ such that (x_i, v_i) has a unique support $((x_i, v_i), (x_j, v_j))$ on c_{ij}, for all $x_k \in X$ linked to both x_i and x_j by a constraint, there exists $v_k \in D(x_k)$ such that $(v_i, v_k) \in c_{ik}$ and $(v_j, v_k) \in c_{jk}$.*

RPC is strictly stronger than AC. An example of a network on which RPC prunes more values than AC is shown in Figure 5. Berlandier proposed an algorithm in $O(end^3)$. The optimal complexity of achieving RPC on a binary normalized network is in $O(en + ed^2 + td^2)$, where t is the number of triples of variables $(x_, , x_j, x_k)$ with c_{ij}, c_{jk} and c_{ik} all in C. An algorithm with this optimal time complexity was presented by Debruyne and Bessiere [42].

In [57], Freuder and Elfe proposed other alternatives to enforce local consistencies stronger than AC whereas modifying only the domains. The idea is to take the inverse of what is done by k-consistency. k-consistency (or $(k-1, 1)$-consistency) ensures that any locally consistent instantiation of size $k-1$ can be extended to any kth variable in a consistent way. This implies the explicit removing of all instantiations of size $k-1$ that cannot fit this property. k-inverse consistency ensures that any locally consistent instantiation of size 1 can be consistently extended to any $k-1$ additional variables. This is $(1, k-1)$-consistency. Since 2-inverse consistency is the same as 2-consistency, the simplest non trivial such inverse consistency is 3-inverse consistency, or path-inverse consistency (PIC), as called in [57].

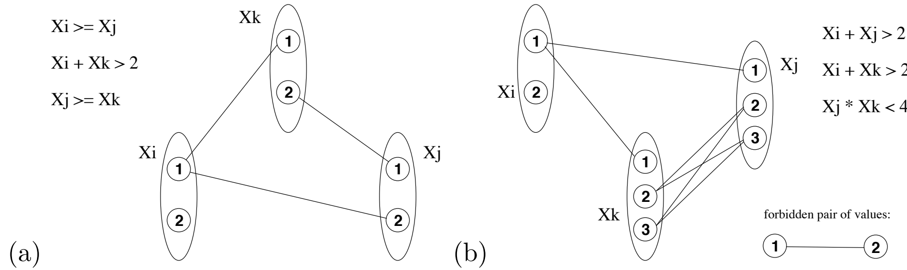

Figure 5: (a) Network on which RPC prunes more than AC: $(x_i, 1)$ is not RPC whereas the whole network is AC. (b) Network on which PIC prunes more than RPC: $(x_i, 1)$ is not PIC whereas the whole network is RPC.

Definition 60 (Path inverse consistency). *A binary normalized network* $N = (X, D, C)$ *is* path-inverse consistent (PIC) *iff for all* $x_i \in X$*, for all* $v_i \in D(x_i)$*, for all* $x_j, x_k \in X$*, there exists* $v_j \in D(x_j)$ *and* $v_k \in D(x_k)$ *such that* $((x_i, v_i), (x_j, v_j), (x_k, v_k))$ *is locally consistent.*

PIC is strictly stronger than RPC. An example of a network on which PIC prunes more values than RPC is shown in Figure 5. Freuder and Elfe proposed an algorithm in $O(en^2d^4)$. In [41], Debruyne proposed some sufficient conditions for the path-inverse consistency of a network. They permit to avoid some constraint checks. Debruyne presented an optimal algorithm for PIC that runs in $O(en + ed^2 + td^3)$.

Following RPC and PIC, Debruyne and Bessiere proposed max-restricted path consistency (maxRPC) [42]. maxRPC still increases the amount of local consistency on triangles of variables. Given a value (x_i, v_i) and a constraint c_{ij}, maxRPC ensures that (x_i, v_i) has a support on c_{ij} path consistent on any third variable.

Definition 61 (Max-restricted path consistency). *A binary normalized network* $N = (X, D, C)$ *is* max-restricted-path consistent (macRPC) *iff for all* $x_i \in X$*, for all* $v_i \in D(x_i)$*, for all* $c_{ij} \in C$*, there exists* $v_j \in D(x_j)$ *such that* $(v_i, v_j) \in c_{ij}$ *and for all* $x_k \in X$ *there exists* $v_k \in D(x_k)$ *with* $((x_i, v_i), (x_j, v_j), (x_k, v_k))$ *locally consistent.*

maxRPC is strictly stronger than PIC. An example of a network on which maxRPC prunes more values than PIC is shown in Figure 6. An optimal algorithm for maxRPC was proposed in [42]. It runs in $O(en + ed^2 + td^3)$.

6.2 Consistency according to the neighborhood

Since k-inverse consistency is polynomial with the exponent depending on k, checking k-inverse consistency is prohibitive if k is large. However, if variables are not uniformly constrained, it can be worthwhile to adapt the level of k-inverse consistency to the size of their neighborhood, focusing filtering effort

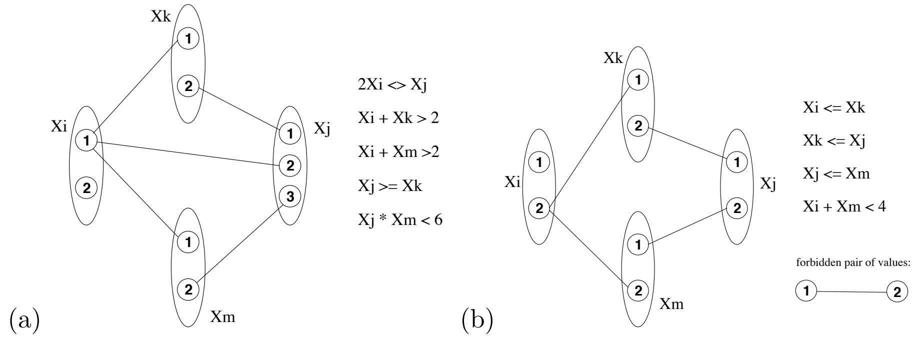

Figure 6: (a) Network on which maxRPC prunes more than PIC: $(x_i, 1)$ is not maxRPC whereas the whole network is PIC. (b) Network on which SAC prunes more than maxRPC: $(x_i, 2)$ is not SAC whereas the whole network is maxRPC.

on the most constrained variables (as it is done in Adaptive consistency — see Section 5.2). This is the basis of neighborhood inverse consistency (NIC, [57]), which ensures that every value v_i in a domain $D(x_i)$ can be extended consistently to all the neighbors of x_i.

Definition 62 (Neighborhood inverse consistency). *A network* $N = (X, D, C)$ *is* neighborhood-inverse consistent (NIC) *iff for all* $x_i \in X$*, for all* $v_i \in D(x_i)$*, the instantiation* (x_i, v_i) *can be extended to a locally consistent instantiation on the set of all variables involved in a constraint with* x_i*.*

An algorithm for NIC was proposed in [57]. It runs in $O(g^2(n + ed)d^{g+1})$, where g is the maximum degree of a variable in the associated hypergraph. It is not proved optimal. Anyway, it seems difficult to go below the exponential factor $nd \cdot d^g$ because every value of every variable must be proved consistent with its neighborhood (possibly of size g). NIC is strictly stronger than maxRPC.

NIC networks do not have the good property of backtrack-free search that adaptive consistent networks have. Although achieving NIC is exponential in the size of the largest neighborhood, it guarantees neither backtrack-free search nor consistency of the network. In addition, the behavior of NIC is dependent on the structure of the network. If two variables x_i and x_j are not neighbors, the network obtained by adding a universal constraint allowing all the pairs of values $(v_i, v_j) \in D(x_i) \times D(x_j)$ between x_i and x_j is equivalent to the initial one. However, as opposed to the other local consistencies, NIC is affected by this change because the neighborhood of x_i has changed. NIC can detect more inconsistent values. Obviously, this process increases time complexity because the sizes of neighborhoods increase.

6.3 Singleton consistencies

A general technique, which has been used in several areas of automated reasoning consists in trying in turn different assignments of a value to a variable, and performing constraint propagation on the subproblem obtained by this assignment. If the problem is found to be inconsistent, this means that this value does not belong to any solution and thus can be pruned. This kind of technique was used on the bounds of interval domains in scheduling ('shaving' in [90]) or on continuous CSPs (3B-consistency in [83]). This technique was also used on literals as a way to derive better variable ordering heuristics in DPLL for SAT formulas (by counting the size of the remaining clauses after instantiation of a literal and unit propagation) in [52, 85]. Finally, it was formalized as a class of local consistencies in [43, 102, 44] under the name 'singleton consistencies'. I give the definition in the case where the amount of propagation applied to each subproblem is arc consistency. Any other local consistency can be used in a similar way. In the following, the subnetwork obtained from a network N by reducing the domain of a variable x_i to the singleton $\{v_i\}$ is denoted by $N|_{x_i=v_i}$.

Definition 63 (Singleton arc consistency). *A network $N = (X, D, C)$ is* singleton arc consistent (SAC) *iff for all $x_i \in X$, for all $v_i \in D(x_i)$, the subproblem $N|_{x_i=v_i}$ is not arc inconsistent.*

SAC is strictly stronger that maxRPC. An example of a network on which SAC prunes more values than maxRPC is given in Fig. 6. The first algorithm for SAC was proposed by Debruyne and Bessiere in [43], and was later named SAC1. It is a brute-force algorithm that checks SAC of each value by performing AC on each subproblem $N|_{x_i=v_i}$. It removes v_i from $D(x_i)$ if $N|_{x_i=v_i}$ is arc inconsistent. After each change in a domain, it rechecks SAC of every remaining value. It can then perform AC nd times on each subproblem, and because there are nd subproblems, it runs in $O(en^2d^4)$ on binary normalized networks, where AC is in $O(ed^2)$. In [6], Bartàk and Erben proposed SAC2, a smarter algorithm that avoids unnecessary work by storing lists of supports, a bit like AC4. Unfortunately, its worst-case time complexity is still $O(en^2d^4)$. Recently, Bessiere and Debruyne showed that the complexity of SAC on binary normalized networks is in $O(end^3)$, and they proposed SAC-Opt, an algorithm with this optimal time complexity [16, 17]. To achieve optimal time, SAC-Opt stores a lot of information in large data structures that require $O(end^2)$ space. SAC-SDS (*Sharing Data Structures*) is a lighter version in which less structures are stored. Its $O(end^4)$ time complexity is a compromise between former SAC algorithms and SAC-Opt, whereas its space complexity is the same as SAC2, namely, $O(n^2d^2)$. Lecoutre and Cardon proposed SAC3, a different technique to enforce SAC [82]. SAC3 incrementally assigns values to variables in the network until arc consistency wipes out a domain. If the current sequence of assignments is $I = ((x_1, v_1), \ldots, (x_k, v_k))$, it deduces that the values $(x_1, v_1), (x_2, v_2), \ldots, (x_{k-1}, v_{k-1})$ are currently SAC. This technique permits to prove SAC of several values in a single arc consistency pass. SAC3 does not have optimal worst-case time complexity but it works well in practice.

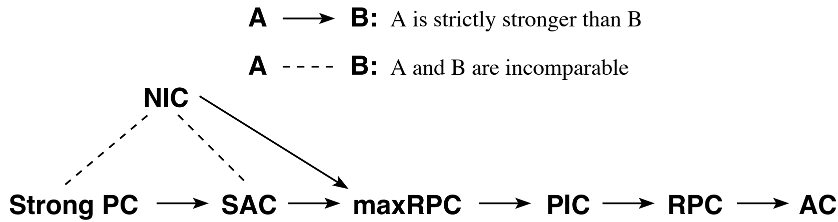

Figure 7: Summary of the comparison between domain-based consistencies lying between AC and Strong PC. $A \rightarrow B$ means that local consistency A is strictly stronger than local consistency B. (The stronger relation is transitive.)

Several extensions of SAC have been proposed. Prosser et al. proposed restricted-SAC, a weakened version of SAC that checks SAC of each value in one pass, without propagating removals to values already processed [102]. Some subtle extensions that are stronger than SAC itself have been proposed in [12, 34, 16]. Their effectiveness and efficiency in practice have not yet been assessed.

Many other singleton consistencies can be constructed because any local consistency can be used to detect the possible inconsistency of the network $N|_{x_i=v_i}$. If a local consistency can be enforced in polynomial time, the corresponding singleton consistency also has a polynomial worst-case time complexity. Prosser et al. analyzed this wider picture. In [102, 123], they theoretically compared the pruning capabilities of (i, j)-consistencies and singleton (i, j)-consistencies:

Theorem 64. *Strong* $(i + 1, j)$*-consistency is strictly stronger than singleton* (i, j)*-consistency. Singleton* $(1, j)$*-consistency is strictly stronger than* $(1, j+1)$*-consistency.*

Fig. 7 summarizes the qualitative comparison between the local consistencies presented in this section. Complete proofs can be found in [44]. Verfaillie et al. proposed a generic algorithm schema that can enforce most of the local consistencies presented in this section, plus new ones that are combinations of existing ones [120].

7 Domain-based Consistencies Weaker than AC

Arc consistency is not the weakest level of consistency we can define on a network. The 80's and first half of 90's have seen quite a lot of works trying to find the amount of filtering that should be performed by a backtrack search procedure. At that time, even if the studies were mostly interested in binary constraints, it was the conventional wisdom that AC was too expensive to be maintained. As a result, several other properties weaker than AC were proposed. The idea behind these properties is to reduce the number of times arc consistency of variables must be checked against constraints. In other words, these properties reduce the number of calls to function `Revise` in a coarse-grained algorithm. They are presented in Section 7.1.

More recently, and essentially because of the cost of arc consistency on non-binary constraints, other forms of consistency were introduced. These techniques do not try to reduce the number of calls to the `Revise` procedure, but instead, they try to reduce the amount of work such a `Revise` procedure performs. Section 7.2 describes them.

7.1 Reducing the number of times constraints are revised

The filtering techniques that try to reduce the number of times constraints are revised are based on properties a network must verify according to some additional parameter such as an ordering on the variables, or a partial instantiation. This extra parameter permits to specify which variables must be arc consistent with which constraints.

7.1.1 Directional arc consistency

Dechter and Pearl proposed directional arc consistency in [47]. The idea is to associate an ordering to the variables in the network and to impose that constraints are arc consistent in the direction of this ordering.

Definition 65 (Directional arc consistency). *A binary network* $N = (X, D, C)$ *is* directional arc consistent (DAC) *according to ordering* $o = (x_{k_1}, \ldots, x_{k_n})$ *on* X*, where* $(k_1, \ldots, k_n)$ *is a permutation of* $(1, \ldots, n)$*, iff for all* $c(x_i, x_j) \in C$*, if* $x_i <_o x_j$ *then* x_i *is arc consistent on* $c(x_i, x_j)$*.*

Directional arc consistency is simpler to enforce than arc consistency. Removing a value v_i in $D(x_i)$ for some variable x_i cannot make a variable x_j directional arc inconsistent on c_{ij} if $x_i <_o x_j$. As a result, there is no need to use a propagation queue for an algorithm achieving DAC. It is sufficient to process the variables from the last in the ordering to the first, revising each variable preceding the current one on the constraint they share, if any (see Algorithm 5).

Algorithm 5: Algorithm for DAC

procedure $DAC(N, o)$;
1 **for** $j \leftarrow n$ ***downto*** 2 **do**
2   **foreach** $c_{ik_j} \in C_N \mid x_i <_o x_{k_j}$ **do**
3   **if** ***not*** `Revise`(x_i, c_{ik_j}) **then** return false

Example 66. Consider the network with variables $X = \{x_1, x_2, x_3\}$, domains $D(x_1) = D(x_2) = \{1..5\}$, $D(x_3) = \{1..3\}$, constraints $C = \{x_1 < x_2, x_2 = x_3, x_1 > x_3\}$, and ordering $o = (x_1, x_2, x_3)$. Revising x_1 on c_{13} and x_2 on c_{23} (in whatever order) prunes value 1 from $D(x_1)$ and values 4 and 5 from $D(x_2)$. Revising x_1 on c_{12} prunes values 3,4,5 from $D(x_1)$. Therefore, $N' = (X, D', C)$ with $D'(x_1) = \{2\}, D'(x_2) = D'(x_3) = \{1, 2, 3\}$ is the DAC closure of N. Note that arc consistency proves inconsistency.

Proposition 67. *The algorithm DAC enforces directional arc consistency according to ordering o in $O(ed^2)$ time.*

7.1.2 Forward checking

Even if they are often presented as stand alone preprocessing of a network, local consistencies are usually intended to be maintained during a backtrack search. It is thus natural to include in our analysis the filtering techniques that were only defined as associated with backtrack search. The amount of filtering performed by the famous *forward checking (FC)* [66, 69] can be defined as a local consistency. The FC search procedure guarantees that at each step of the search, all the constraints between already assigned variables and not yet assigned variables are arc consistent.

Definition 68 (Forward checking). *Let $N = (X, D, C)$ be a binary network and $Y \subseteq X$ such that $|D(x_i)| = 1$ for all $x_i \in Y$. N is* forward checking consistent (FC) *according to the instantiation I on Y iff I is locally consistent and for all $x_i \in Y$, for all $x_j \in X \setminus Y$, for all $c(x_i, x_j) \in C$, x_j is arc consistent on $c(x_i, x_j)$.*

Algorithm 6 presents a procedure that applies FC on a network N according to a subset of instantiated variables $Y \cup \{x_i\}$ if N is already FC according to Y.

Algorithm 6: Algorithm for FC

procedure $FC(N, Y, x_i)$;
1 **foreach** $c_{ij} \in C_N \mid x_j \in X \setminus Y$ **do**
2   **if** ***not*** `Revise`(x_j, c_{ij}) **then** return false

Example 69. On the network of Example 66, where $X = \{x_1, x_2, x_3\}$, domains $D(x_1) = D(x_2) = \{1..5\}$, $D(x_3) = \{1..3\}$, and $C = \{x_1 < x_2, x_2 = x_3, x_1 > x_3\}$, applying FC according to $\{x_1\}$ after an instantiation $x_1 = 3$ (i.e., $D(x_1) = \{3\}$) prunes values 1,2,3 from $D(x_2)$ and 3 from $D(x_3)$.

FC has the property that once a variable x_j is made arc consistent on c_{ij} ($|D(x_i)| = 1$), it remains AC on c_{ij} in spite of any future domain reduction, because x_i is singleton. This means that each constraint needs to be revised only once along a branch of instantiations. As opposed to chronological backtracking, a procedure maintaining FC does not need to check consistency of values of the current variable against already instantiated ones. FC is the weakest level of local consistency with this property.

The complexity of a call to `Revise` in FC is in $O(d)$ because one of the domains involved is a singleton. Hence, enforcing FC on a binary network according to a partial instantiation of arbitrary length is in $O(ed)$.

The definition of FC can be extended to non-binary constraints in several different ways. Van Hentenryck proposed a basic one in [116]: A network is

Algorithm 7: Algorithms for PL and FL

```
procedure PL(N, Y, x_i);
1 FC(N, Y, x_i);
2 foreach j ← i+1 to n do
3     foreach k ← j+1 to n | c_jk ∈ C_N do
4         if not Revise(x_j, c_jk) then return false

procedure FL(N, Y, x_i);
5 FC(N, Y, x_i);
6 foreach j ← i+1 to n do
7     foreach k ← i+1 to n, k ≠ j | c_jk ∈ C_N do
8         if not Revise(x_j, c_jk) then return false
```

FC according to a partial instantiation I on a subset Y of X if and only if I is locally consistent and for all $x_j \in X \setminus Y$, for all $c \in C$ such that $X(c) \setminus Y = \{x_j\}$, x_j is arc consistent on c. Bessiere et al. presented five additional extensions of FC to non-binary constraints [22].

7.1.3 Other lookahead filterings

The idea of reducing the amount of filtering of arc consistency to avoid complex algorithms led to other forms of propagation. In [69], Haralick and Elliott proposed *partial lookahead (PL)* and *full lookahead (FL)*, two levels of propagation, stronger that FC. Haralick and Elliott gave operational definitions of PL and FL in terms of algorithms performing a given amount of filtering. As opposed to DAC and FC, no clear property on the output of PL or FL can be specified and thus, no clear fixpoint can be defined.

PL and FL are presented in Algorithm 7. Given a network N, an ordering $o = (x_1, \ldots, x_n)$, and a current variable x_i, PL first performs FC and then takes the variables x_j from x_{i+1} to x_n and calls `Revise` for x_j on each $c_{jk}, j < k \leq n$. FL performs a stronger level of filtering than PL. Given a network N, an ordering o, and a current variable x_i, FL takes the variables x_j from x_{i+1} to x_n and calls `Revise` for x_j on each $c_{jk}, i < k \leq n, k \neq j$.

PL and FL cannot guarantee any property on the arc consistency of arcs at the end of the process. After x_j has been made arc consistent on $c_{jk}, j < k$, values of x_k can be removed when making x_k arc consistency on arcs leaving x_k. Thus, it is no longer guaranteed that the arc (x_j, c_{jk}) is arc consistent at the end of the process because each arc is revised only once.

The complexity of function `Revise` is in $O(d^2)$ because it is called on constraints involving non singleton variables for both PL and FL. Thus, PL and FL are in $O(ed^2)$, like DAC.

In [98, 99], Nadel encapsulated these forms of consistency in a general schema, going from the consistency maintained by simple backtracking, i.e., local consistency of the instantiated variables (noted $\mathrm{AC}_{1/5}$), to arc consistency. FC is denoted by $\mathrm{AC}_{1/4}$ while PL and FL are denoted by $\mathrm{AC}_{1/3}$ and $\mathrm{AC}_{1/2}$ respectively.

In [113], Tsang gave a comparison of the pruning capabilities of these different levels of filtering. He proved that DAC and FL are strictly stronger than PL, which itself is strictly stronger than FC. FL and DAC are incomparable: There are cases where DAC prunes values FL does not prune and vice versa. AC is strictly stronger than all of them.

7.1.4 Selective revision

As a last technique to reduce the number of calls to function `Revise` in arc consistency, there is the work by Freuder and Wallace. They proposed to use criteria to discard arc revisions when they are not likely to be effective [58].

Given a coarse-grained arc consistency algorithm, *distance-bounded* propagation confines constraint propagation to a fixed distance δ from the variables at which it began. This is implemented by attaching a *stamp* to each arc in Q. Arcs put in Q in the initialization of the arc consistency call are stamped with zero. Forthcoming arcs are stamped with $t + 1$ if t is the stamp of the revised arc that provoked their addition to Q. When an arc is to be stamped with a value greater than the maximal distance δ, it is not put in Q.

Response-bounded propagation stops subsequent propagations when the amount of change in a domain falls below a given threshold r. This is implemented by testing if the ratio of values removed by a revision is greater than r before adding relevant arcs in Q.

7.2 Using the order on the domains to relax `Revise`

The second form of local consistencies weaker than arc consistency do not try to reduce the number of arc revisions, but instead, they try to reduce the cost of revisions to overcome the prohibitive cost of generalized arc consistency on some constraints. The idea behind these local consistencies is to use the fact that domains are composed of integers. Integer domains inherit the total ordering on $\mathbb{Z}$ and by consequence they inherit the two particular values $min_D(x_i)$ and $max_D(x_i)$, called the *bounds* of $D(x_i)$. I present two ways of relaxing generalized arc consistency on a constraint c. The first option is to ensure support on c only for the bounds of the domain of each variable in $X(c)$. The second option is to look for supports not in $\pi_{X(c)}(D)$ but in $\pi_{X(c)}(D^I)$, where D^I is the domain such that for all $x_i, D^I(x_i) = \{v \in \mathbb{Z} \mid min_D(x_i) \leq v \leq max_D(x_i)\}$. Using the first option or the second, or combining both, give rise to three relaxed forms of local consistency.

Definition 70 (Consistencies on bounds). *Given a network $N = (X, D, C)$, given a constraint c, a* bound support *τ on c is a tuple that satisfies c and such that for all $x_i \in X(c), min_D(x_i) \leq \tau[x_i] \leq max_D(x_i)$, that is, $\tau \in c \cap \pi_{X(c)}(D^I)$. (A bound support in which each variable is assigned a value in its domain is a support.)*

- *A constraint c is* bound(Z) consistent (BC(Z)) *iff for all $x_i \in X(c)$, $(x_i, min_D(x_i))$ and $(x_i, max_D(x_i))$ belong to a bound support on c.*

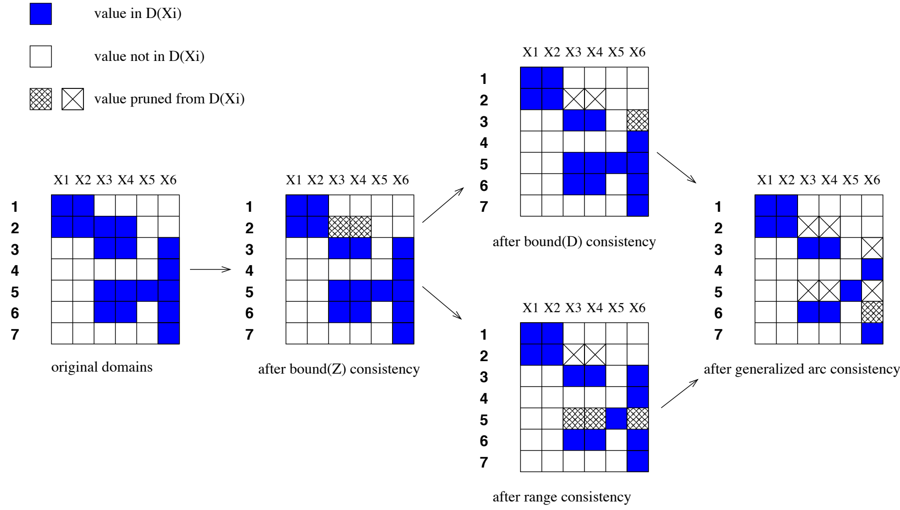

Figure 8: Amount of propagation performed by BC(Z), BC(D), RC and GAC on the constraint `alldifferent`$(x_1 \dots, x_6)$ of Example 71.

- *A constraint c is* range consistent (RC) *iff for all $x_i \in X(c)$, for all $v_i \in D(x_i)$, (x_i, v_i) belongs to a bound support on c.*
- *A constraint c is* bound(D) consistent (BC(D)) *iff for all $x_i \in X(c)$, $(x_i, min_D(x_i))$ and $(x_i, max_D(x_i))$ belong to a support on c.*

The network N is bound(Z) / range / bound(D) consistent iff all its constraints are bound(Z) / range / bound(D) consistent.

Example 71. Consider the network with variables $x_1, \dots, x_6$, domains $D(x_1) = D(x_2) = \{1, 2\}$, $D(x_3) = D(x_4) = \{2, 3, 5, 6\}$, $D(x_5) = \{5\}$, $D(x_6) = [3..7]$ and $C = \{$`alldifferent`$(x_1 \dots, x_6)\}$. These domains are depicted in Fig. 8 after BC(Z), BC(D), RC, and GAC are applied to the constraint.

The notion of local consistency on bounds comes from works on arithmetic constraints over real variables (variables taking values in intervals of reals). The move to integer is not so direct, and the names chosen in Definition 70 are not unanimously used in the literature.[8] Collavizza et al. presented a review of several local consistency notions for continuous domains [36]. One of these local consistencies is named bound consistency but has no link at all with those named bound(Z) or bound(D) consistencies in Definition 70. In [110, 2], Schulte and Stuckey, and Apt gave a definition for bound consistency on integer variables that is the direct application of the definition on reals. Choi et al. called it *bound(R) consistency* [35]. Bound(R) consistency differs from bound(Z) consistency in that it looks for bound supports composed of *real* values

[8]We took the names bound(Z) consistency and bound(D) consistency in [35] and range consistency in [80].

(instead of integer values). For instance, in Fig. 8 the bound $(x_3, 2)$, removed by BC(Z), is bound(R) consistent because it belongs to the tuple $(1, \frac{3}{2}, 2, 4, 5, 3)$ which satisfies the `alldifferent` constraint and where each variable takes a real value between its bounds. In [110], Schulte and Stuckey showed that there exist constraints on which BC(R) is polynomial to enforce whereas BC(Z) is NP-hard. The bound consistency of Dechter in [46] corresponds to bound(D) consistency in Definition 70. The local consistency named interval consistency by Van Hentenryck et al. and by Apt in [119, 2] corresponds to bound(Z) consistency in Definition 70. In recent papers dealing with integer variables, it seems that the name bound consistency is uniformly used to refer to BC(Z) [103, 105]. As seen in Fig. 8, these local consistencies do not all prune the same amount of values.

Theorem 72. *Generalized arc consistency is strictly stronger than range and bound(D) consistencies, which are themselves strictly stronger than bound(Z) consistency, which itself is strictly stronger than bound(R) consistency. Bound(D) consistency and range consistency are incomparable.*

These local consistencies being all strictly weaker than GAC, the only reason to use one of them instead of GAC is to have a faster algorithm. BC(D) requires finding supports, as in GAC. Hence, it only decreases the cost by a factor d because it seeks supports for 2 values per domain (the bounds) instead of d values. RC, BC(Z) and BC(R) look for bound supports (on integers or on reals). Now, looking for bound supports is not necessarily simpler than looking for supports. To keep things simple, let us focus on BC(Z) but the same reasoning applies to BC(R) or RC.

Proposition 73 (Complexity of bound(Z) consistency). *Deciding bound(Z) consistency of a constraint can take exponential time, even if the constraint is binary, where arc consistency is in $O(d^2)$ time, d being the size of the largest domain.*

Proof. Let c be any binary constraint with no particular semantics that could be used by a propagation algorithm. It is well-known that deciding AC on such a constraint is in $O(d^2)$ time [92]. If $X(c) = (x_i, x_j)$, deciding BC(Z) is done by looking for a bound support for the four values $min_D(x_i)$, $min_D(x_j)$, $max_D(x_i)$, $max_D(x_j)$. Finding a bound support for $min_D(x_i)$ is done by exploring $D^I(x_j)$. Suppose c is characterized by a Boolean function requiring constant space (this is often the case), and suppose $D(x_i) = \{min_D(x_i), max_D(x_i)\}$, $D(x_j) = \{min_D(x_j), max_D(x_j)\}$, with $max_D(x_i) < max_D(x_j)$. The size of the input is in $O(log_2(max_D(x_j)))$. The cost of exploring the whole set $D^I(x_j)$ is thus exponential in the size of the input. □

A direct consequence of Proposition 73 is that deciding BC(Z) is NP-hard. Suppose φ is a set of clauses on the Boolean variables $x_1, \ldots, x_n$ and $c(y,z)$ is the constraint satisfied by all tuples on (y, z) where $y \neq 0$ if and only if the bit vector of size n representing $z \ mod \ 2^n$ in base 2 is a tuple of 0/1 values for

$x_1, \ldots, x_n$ that satisfies φ. If $max_D(z) - min_D(z) \geq 2^n$, deciding BC(Z) for a bound of $D(y)$ other than 0 is equivalent to deciding the satisfiability of φ.

This shows that bound(Z) consistency is useful only if the constraint we want to propagate has inherent properties that permit a computation of bound supports faster than supports. Take for example the constraint $\texttt{sum}_k(x_1, \ldots, x_n)$ that holds if and only if $\sum_{i=1}^{n} x_i = k$. Deciding generalized arc consistency on this constraint is NP-complete because we can easily transform the decision problem SubsetSum [60] into the problem of deciding whether a $\texttt{sum}_k$ constraint has support.[9] On the contrary, testing bound(Z) consistency on the $\texttt{sum}_k$ constraint is polynomial because it is sufficient to verify that $min_D(x_i)$ is at least $k - \sum_{j \neq i} max_D(x_j)$ and $max_D(x_i)$ is at most $k - \sum_{j \neq i} min_D(x_j)$, for all $i, 1 \leq i \leq n$.

Zhang and Yap showed that bound(R) consistency is equivalent to generalized arc consistency when constraints are linear [126]. For example, the constraint $\sum_{i=1}^{n} x_i \leq k$ is bound(R) consistent if and only if it is generalized arc consistent. Schulte and Stuckey gave other sufficient conditions on constraints that permit to guarantee that bound(R) consistency is equivalent to generalized arc consistency [110]. Even when a constraint does not fit the required conditions, there are cases where additional properties on the input domains guarantee that the output of enforcing BC(R) will be GAC. For instance, if all variables have interval domains, BC(R) on the $\texttt{sum}_k$ constraint is equivalent to GAC. This is a polynomial case for GAC on $\texttt{sum}_k$, which is NP-hard in general. Interestingly, Schulte and Stuckey showed how to analyze a constraint model to discover on which constraints generalized arc consistency enforcing can be replaced by bound(R) consistency whereas preserving the amount of pruning.

8 Constraint Propagation as Iteration of Reduction Rules

Local consistency is a way to formally define which amount of consistency we want a network to guarantee, and as a consequence, which network will be produced by an algorithm enforcing this level of consistency. But nothing is said about the way the algorithm enforces it. Rules iteration takes the question on the other side. A reduction rule specifies under which conditions and on which constraints operations of filtering are performed. The network produced guarantees a formal property such as a given level of local consistency only if the reduction rules and the way they are applied have some good properties. The rules iteration approach was first formalized by Montanari and Rossi under the name *relaxation rules* [96]. Benhamou et al. studied rules iteration *via* interval arithmetics (that is, reducing only bounds) [10, 11]. Constraint Handling Rules (CHR) is a programming language based on reduction rules (see [59] and Chapter 13 in Part II of [109]). In [1, 2], Apt gave a comprehensive presentation

[9] The constraint $\texttt{sum}_k(x_1, \ldots, x_n)$ where $D(x_i) = \{0, j_i\}$ has support if and only if there exists a subset of $\{j_1, \ldots, j_n\}$ of sum k, which is exactly the SubsetSum problem.

of the rules iteration approach. I essentially follow Apt's presentation of the concept of reduction rule.

A reduction rule is simply a function that maps a network to another, where the image is a tightening of the input.

Definition 74 (Reduction rule). *Given a network N, a* reduction rule *is a function f from $\mathcal{P}_N$ to $\mathcal{P}_N$ such that for all $N' \in \mathcal{P}_N, f(N') \in \mathcal{P}_{N'}$.*

We should bear in mind that $\mathcal{P}_N$ contains all the networks that are tightenings of N (see Definition 13). In most cases, reduction rules are reduction steps that reduce a single variable domain according to a single constraint. I name them propagators.

Definition 75 (Propagator). *Given a constraint c in a network $N_c = (X, D, \{c\})$, a* propagator *f for c is a reduction rule from $\mathcal{P}_{N_c}$ to $\mathcal{P}_{N_c}$ that tightens only domains independently of the constraints other than c. That is, for all $N' = (X, D', C') \in \mathcal{P}_{N_c}$, $f(N') = (X, D'', C')$, with $D'' \subseteq D'$ and $D'' = D_{f(X,D',\{c\})}$.*

Propagators can verify some properties.

Definition 76 (Properties of propagators). *Given a network $N = (X, D, C)$ and two propagators f and g on $\mathcal{P}_{ND}$:*

- *f is called* monotonic *if $N_1 \leq N_2$ implies $f(N_1) \leq f(N_2)$ for all $N_1, N_2 \in \mathcal{P}_{ND}$,*
- *f is called* idempotent *if $ff(N_1) = f(N_1)$ for all $N_1 \in \mathcal{P}_{ND}$,*
- *we say that f and g* commute *if $fg(N_1) = gf(N_1)$ for all $N_1 \in \mathcal{P}_{ND}$,*

I give examples of propagators that do not verify these properties.

Example 77. Consider two networks $N_1 = (X, D_1, C)$ and $N_2 = (X, D_2, C)$ with $C = \{c \equiv (x_1 = x_2)\}$, $D_1(x_1) = \{1, 2\}$, $D_1(x_2) = \{2\}$, $D_2(x_1) = \{1, 2, 3\}$ and $D_2(x_2) = \{2\}$. Consider the propagator f that prunes all values from x_1 that have no support on c if less than half of them have support. f is *not* monotonic because $D_{f(N_1)} \not\subseteq D_{f(N_2)}$ whereas $D_{N_1} \subseteq D_{N_2}$ (f reduces $D_2(x_1)$ to $\{2\}$). Consider the propagator g that prunes *one of* the values from x_1 that have no support on c if such a value exists. g is *not* idempotent because $D_{gg(N_2)} \neq D_{g(N_2)}$ (g reduces $D_2(x_1)$ to $\{1, 2\}$ or $\{2, 3\}$ whereas gg reduces it to $\{2\}$). f and g do *not* commute because $D_{fg(N_2)} \neq D_{gf(N_2)}$ (fg reduces $D_2(x_1)$ to $\{1, 2\}$ or $\{2, 3\}$ whereas gf reduces it to $\{2\}$).

Most of the propagators used in practice satisfy the properties of Definition 76. Among them, monotonicity is particularly interesting. I first need to define what I mean by iteration and by stability of a propagator.

Definition 78 (Iteration). *Let $N = (X, D, C)$ be a network and $F = \{f_1, \ldots, f_k\}$ be a finite set of propagators on $\mathcal{P}_{ND}$. An* iteration *of F on N is a sequence $\langle N_0, N_1, \ldots \rangle$ of elements of $\mathcal{P}_{ND}$ defined by*

$$N_0 = N,$$

$$N_j = f_{n_j}(N_{j-1}),$$

where $j > 0$ and $n_j \in [1..k]$. We say that f_{n_j} is activated *at step j.*

Definition 79 (Stability). *Let $N = (X, D, C)$ be a network and F be a set of propagators on $\mathcal{P}_{ND}$. A network $N' \in \mathcal{P}_{ND}$ is* stable *for F iff for all $f \in F, f(N') = N'$.*

There can be many networks in $\mathcal{P}_{ND}$ that are stable for a given set of propagators. But monotonicity of propagators implies that only one of them will be produced.

Proposition 80 (Least fixpoint). *Let $N = (X, D, C)$ be a network and F be a set of propagators on $\mathcal{P}_{ND}$. If $S = \langle N_0, N_1, \dots \rangle$ is an infinite iteration of F where each $f \in F$ is activated infinitely often, then there exists $j \geq 0$ such that N_j is stable for F. If all f in F are monotonic, N_j is unique and is called the* least fixpoint *of F on N.*

Algorithm 8 is a procedure that takes as input a network N and a set F of propagators on $\mathcal{P}_{ND}$. Thanks to Proposition 80, we are guaranteed that it terminates. If all f in F are monotonic, the output of Algorithm 8 is the least fixpoint of F on N.

Algorithm 8: Generic Iteration Algorithm

```
procedure Generic-Iteration(N, F);
  G ← F;
  while G ≠ ∅ do
    select and remove g from G;
    if N ≠ g(N) then
      update(G);
      N ← g(N);
  /* update(G) adds to G at least all functions f in F \ G for which
  g(N) ≠ f(g(N)) */
```

Sometimes, in addition to monotonicity, propagators can have some other properties. In those cases, Algorithm 8 can be simplified, while still ensuring to produce the same result.

Proposition 81 (Direct iteration). *Let $N = (X, D, C)$ be a network and $F = \{f_1, \dots, f_k\}$ be a set of monotonic and idempotent propagators on $\mathcal{P}_{ND}$ that commute with each other. If an iteration $S = \langle N_0, N_1, \dots, N_k \rangle$ is such that $N = N_0$ and for all $f_i \in F$ there exists $N_j \in S$ such that $N_j = f_i(N_{j-1})$, then N_k is stable for F and is the least fixpoint of F on N.*

Proposition 81 guarantees that Algorithm 9 produces the least fixpoint of F.

By defining the appropriate set of propagators, we can obtain most of the local consistencies presented in previous sections. For instance, we can enforce

Algorithm 9: Direct Iteration Algorithm

```
procedure Direct-Iteration(N, F);
   G ← F;
   while G ≠ ∅ do
      select and remove g from G;
      N ← g(N);
```

arc consistency on a network $N = (X, D, C)$. I first define the propagators $f_{i,j}$ such that:

$$\forall N_1 = (X, D_1, C) \in \mathcal{P}_{ND}, \forall x_i \in X, \forall c_j \in C,\ f_{i,j}(N_1) = (X, D'_1, C) \text{ with}$$

$$D'_1(x_i) = \pi_{\{x_i\}}(c_j \cap \pi_{X(c_j)}(D_1)) \text{ and } D'_1(x_k) = D_1(x_k), \forall k \neq i.$$

I consider the set of propagators $F_{AC} = \{f_{i,j} \mid x_i \in X, c_j \in C\}$. They are all monotonic. Then, *Generic-Iteration*(N, F_{AC}) terminates on the least fixpoint for F_{AC}, which is the arc consistent closure of N.

It is shown in [2] that we can also enforce higher-order consistencies, such as path consistency, by defining sets of monotonic propagators that involve several constraints at a time and that alter the set of constraints.

9 Specific Constraints

In previous sections, I presented constraint propagation and local consistencies in a generic way without saying what should be done when we have some specific information on the semantics of a constraint. In this section, I develop some of the available techniques to take into account constraint semantics.

9.1 Specific propagators in solvers

All constraint solvers attach a specific propagation algorithm to the specific types of constraints they contain. In addition, most of them allow the user to design her own propagators for the new constraints she incorporates. The fact that arithmetic constraints are at the core of most constraint solvers influences the way these solvers are implemented. Not only all basic arithmetic constraints are present, but the programming possibilities they provide for building new propagators is arithmetic-oriented. I give a brief overview of what is the common point to most solvers. The art of designing constraint propagators is not a mature science yet, and things can differ from one solver to another, and will most probably evolve in the next years. This topic has been addressed in some academic publications [79, 94, 118, 119, 78, 111] and in manuals of constraint solvers. See also Chapter 14 in Part II of [109].

In most arithmetic constraints, it appears that a reduction of a domain does not produce the same effect on the other variables of the constraint, depending on if it is the removal of a value in the middle of the domain, if it is the increase

Algorithm 10: AC3-like constraint propagation schema

```
function Constraint-Propag(in X: set): Boolean ;
   begin
1     foreach c ∈ C do perform init-propag on c and update Q with relevant
      events;
2     while Q ≠ ∅ do
3        select and remove (x_i, c, x_j, Mtype) from Q;
4        if Revise(x_i, c, (x_j, Mtype), Changes) then
5           if D(x_i) = ∅ then return false ;
6           foreach c' ∈ Γ^C(x_i), Mtype ∈ Changes do
7              foreach x_j ∈ X(c'), j ≠ i do Q ← Q ∪ {(x_j, c', x_i, Mtype)};
8     return true ;
   end
   /* Γ^C(x_i) is the set of constraints with x_i in their scheme */
```

of its minimum value, if it is the decrease of its maximum value, or if it is an instantiation to a single value. Then, it is worth differentiating these different types of *events* to be able to propagate exactly as necessary. The events usually recognized by constraint solvers are:

- `RemValue`: when a value v is removed from $D(x_i)$
- `IncMin`: when the minimum value of $D(x_i)$ increases
- `DecMax`: when the maximum value of $D(x_i)$ decreases
- `Instantiate`: when $D(x_i)$ becomes a singleton

The way these events are used in a constraint solver is usually bound to the type of propagation architecture handled by the solver. The description I give here is just an illustrative example of how to use those events. If we follow an AC3 like schema of propagation, the use of event types leads to a modified version of Algorithm 1 that takes into account the type *Mtype* of reduction performed on a domain (see Algorithm 10). The modified function `Revise` has parameters $(x_i, c, (x_j, Mtype), Changes)$ where (x_i, c) is the arc to revise because of an *Mtype* change in $D(x_j)$. In addition to a Boolean indicating if a domain has been changed, function `Revise` returns the set *Changes* of the types of changes it performed on $D(x_i)$ (line 4). Each modification of type *Mtype* on domain $D(x_i)$ requires the addition of 4-tuples $(x_j, c', x_i, Mtype)$ to the list Q of pending events (lines 6–7). The presence of $(x_j, c, x_i, Mtype)$ in Q means that x_j should be revised on c because of an *Mtype* change in $D(x_i)$. I suppose that each constraint is associated with a function `init-propag` that performs the very first pass of propagation on the constraint and appends to list Q all 4-tuples relevant to events performed on some domains (line 1).

The benefit of this differentiation between types of events is twofold. First, it permits to process constraint propagation differently according to the type of event (line 4). As shown in the following example, this can have a dramatic effect on the cost of revision.

Example 82. Let $x_1 \leq x_2$, with $D(x_1) = D(x_2) = \{1..100\}$. If value 100 is removed from $D(x_2)$, the regular `Revise` procedure of AC3 takes each of the 100 values in $D(x_1)$ one by one, and looks for a support by traversing $D(x_2)$. This requires $1+2+\ldots+99+99 = \frac{100 \cdot 101}{2} - 1$ constraint checks to discover that $(x_1, 100)$ must be removed. An adapted `Revise` procedure knowing that 100 is a `DecMax` event simply decreases $max_D(x_1)$ to the same value as $max_D(x_2)$, i.e., 99. If the value removed from $D(x_2)$ is 50, again regular `Revise` performs around 5,000 constraint checks whereas a specific `Revise` knows that removing 50 is a `RemValue` event for which nothing should be done because the only events that can alter $D(x_1)$ are `DecMax` and `Instantiate`. Algorithm 11 is a specific function `Revise` for constraints $x_{k_1} \leq x_{k_2}$ (k_l is the index of the lth variable in the scheme of the constraint).

Algorithm 11: Function `Revise` for the constraint of Example 82

function `revise`(**inout** x_i; **in** $c \equiv x_{k_1} \leq x_{k_2}$; **in** $(x_j, Mtype)$; **out** $Changes$)**:**
  Boolean ;
  $Changes \leftarrow \emptyset$;
  switch $Mtype$ **do**
    case *RemValue*
      nothing;
    case *IncMin*
      if $j = k_1$ **then** remove all $v < min_D(x_j)$ from $D(x_i)$;
    case *DecMax*
      if $j = k_2$ **then** remove all $v > max_D(x_j)$ from $D(x_i)$;
    case *Instantiate*
      if $j = k_1$ **then** remove all $v < min_D(x_j)$ from $D(x_i)$;
      else remove all $v > max_D(x_j)$ from $D(x_i)$;
  $Changes \leftarrow$ the types of changes performed on $D(x_i)$;

The second advantage of the information on events is that in some cases, we know that it is useless to propagate a constraint because a given event cannot alter the other variables of the constraint. For instance, in the constraint $x_1 \leq x_2$ of the example above, `RemValue` has no effect. Instead of having a set $\Gamma^C(x_i)$ of all constraints involving x_i, we can build such a set for each type of event. $\Gamma^C_{Mtype}(x_i)$ only contains constraints involving x_i for which an $Mtype$ event on x_i requires propagation. Line 6 in Algorithm 10 becomes:

6 **foreach** $c' \in \Gamma^C_{Mtype}(x_i), Mtype \in Changes$ **do** ...

Example 83. Let $c \equiv x_1 \leq x_2$. The only events that require propagation are `IncMin` and `Instantiate` on x_1, and `DecMax` and `Instantiate` on x_2. Thus, c is only put in $\Gamma^C_{IncMin}(x_1)$, $\Gamma^C_{Instantiate}(x_1)$, $\Gamma^C_{DecMax}(x_2)$, and $\Gamma^C_{Instantiate}(x_2)$. It avoids not only useless calls to `Revise` but also insertions and deletions of useless events in Q.

In the extreme case, the domains have been reduced in such a way that a constraint c is entailed. That is, c is satisfied for any valid combination of values

on $X(c)$. (See [119, 111] or Section 4.1.) c can then be removed from the set of constraints of the network as long as the domains are not relaxed.

Example 84. Let $c \equiv x_1 \leq x_2$, $D(x_1) = \{1, 2, 4\}$ and $D(x_2) = \{5, 6, 7\}$. Any valid instantiation of x_1 and x_2 satisfies c. So, c can safely be removed from the network.

There is a third way of saving work during the propagation of changes in domains. It consists in storing not only the type of change performed on a domain $D(x_i)$ during a call to `Revise`, but also the set Δ_i of values removed. Function `Revise` has the extra parameter Δ_j of the values removed from $D(x_j)$ that led to this revision. In addition to *Changes*, `Revise` returns the set Δ_i of values it removes from $D(x_i)$. Δ_i is put in Q with the other information. Lines 3–7 in Algorithm 10 become:

3 select and remove $(x_i, c, x_j, Mtype, \Delta_j)$ from Q;
4 **if** `Revise`$(x_i, c, (x_j, Mtype, \Delta_j), Changes, \Delta_i)$ **then**
5 **if** $D(x_i) = \emptyset$ **then** return **false**;
6 **foreach** $c' \in \Gamma^C_{Mtype}(x_i), Mtype \in Changes$ **do**
7 **foreach** $x_j \in X(c'), j \neq i$ **do** $Q \leftarrow Q \cup \{(x_j, c', x_i, Mtype, \Delta_i)\}$

Such a facility was already proposed by Van Hentenryck et al. in the AC5 propagation schema [118]. This notably permits to decrease the complexity of arc consistency on functional or anti-functional constraints.

Example 85. The functional constraint $x_{k_1} = x_{k_2} + m$ can be propagated by the function `Revise` in Algorithm 12.

Algorithm 12: Function `Revise` for the constraint of Example 85

function `revise`**(inout** x_i; **in** $c \equiv x_{k_1}{=}x_{k_2} + m$; **in** $(x_j, Mtype, \Delta_j)$);
out *Changes*; **out** Δ_i**): Boolean** ;
Changes $\leftarrow \emptyset$;
switch *Mtype* **do**
case *`RemValue`*
if $j = k_1$ **then** **foreach** $v \in \Delta_j$ **do** remove $(v - m)$ from $D(x_i)$;
else **foreach** $v \in \Delta_j$ **do** remove $(v + m)$ from $D(x_i)$;
case *`IncMin`*
if $j = k_1$ **then** remove all $v < min_D(x_j) - m$ from $D(x_i)$;
else remove all $v < min_D(x_j) + m$ from $D(x_i)$;
case *`DecMax`*
if $j = k_1$ **then** remove all $v > max_D(x_j) - m$ from $D(x_i)$;
else remove all $v > max_D(x_j) + m$ from $D(x_i)$;
case *`Instantiate`*
if $j = k_1$ **then** assign $min_D(x_j) - m$ to x_i;
else assign $min_D(x_j) + m$ to x_i;
Changes $\leftarrow$ the types of changes performed;
$\Delta_i \leftarrow$ all values removed from $D(x_i)$;

These four types of events permit to build efficient propagators for elementary constraints. But as soon as constraints are not arithmetic or do not have properties such as being functional, antifunctional or others, it is difficult to implement propagators with this kind of architecture.

9.2 Classes of specific constraints: global constraints

There are 'constraint patterns' that are ubiquitous when trying to express real problems as constraint networks. For example, we often need to say that a set of variables must all take different values. The size of the pattern is not fixed, that is, there can be any number of variables in the set. The `alldifferent` constraint, as introduced in CHIP [50], is not a single constraint but a whole class of constraints. Any constraint specifying that its variables must all take different values is an `alldifferent` constraint. The conventional wisdom is to name 'global constraints' these classes of constraints defined by a Boolean function whose domain contains tuples of values of any length. An instance c of a given global constraint is a constraint with a fixed scheme of variables which contains all tuples of length $|X(c)|$ accepted by the function defining the global constraint.[10] In the last years, the literature became quite verbose on this subject. Beldiceanu et al. proposed an extensive list of global constraints [9].

Example 86. The $\texttt{alldifferent}(x_1, \ldots, x_n)$ global constraint is the class of constraints that are defined on any sequence of n variables, $n \geq 2$, such that $x_i \neq x_j$ for all $i, j, 1 \leq i, j \leq n, i \neq j$. The $\texttt{NValue}(y, [x_1, \ldots, x_n])$ global constraint is the class of constraints that are defined on any sequence of $n+1$ variables, $n \geq 1$, such that $|\{x_i \mid 1 \leq i \leq n\}| = y$ [100, 8].

It is interesting to incorporate global constraints in constraint solvers so that users can use them to express the corresponding constraint pattern easily. Because these global constraints can be used with a scheme of any size, it is important to have a way to propagate them without using generic arc consistency algorithms. (Remember that optimal generic arc consistency algorithms are in $O(erd^r)$ for constraints involving r variables —see Section 4.1.)

The first alternative to the combinatorial explosion of generic algorithms for GAC on a global constraint is to decompose it with 'simpler' constraints. A *decomposition* of a global constraint G is a polynomial time transformation δ_k (k being an integer) that, given any network $N = (X(c), D, \{c\})$ where c is an instance of G, returns a network $\delta_k(N)$ such that $X(c) \subseteq X_{\delta_k(N)}$, for all $x_i \in X(c), D(x_i) = D_{\delta_k(N)}(x_i)$, for all $c_j \in C_{\delta_k(N)}$, $|X(c_j)| \leq k$, and $sol(N) = \pi_{X(c)}(sol(\delta_k(N)))$. That is, transforming N in $\delta_k(N)$ means replacing c by some new bounded arity constraints (and possibly new variables) whereas preserving

[10] This definition does not allow constraints defined on several sequences of variables, such as the $\texttt{disjoint}([x_1 \ldots, x_n], [y_1 \ldots, y_m])$ constraint [9]. In such a case, we need to extend to Boolean functions with parameters giving the length of each sequence. This is essentially the same.

the set of tuples allowed on $X(c)$. Note that by definition, the domains of the additional variables in the decomposition are necessarily of polynomial size.[11]

Example 87. The global constraint $\texttt{atmost}_{p,v}(x_1, \ldots, x_n)$ holds if and only if at most p variables in $x_1, \ldots, x_n$ take value v [117]. This constraint can be decomposed with $n+1$ additional variables $y_0, \ldots, y_n$. The transformation involves the constraint $(x_i = v \wedge y_i = y_{i-1} + 1) \vee (x_i \neq v \wedge y_i = y_{i-1})$ for all $i, 1 \leq i \leq n$, and the domains $D(y_0) = \{0\}$ and $D(y_i) = \{0, \ldots, p\}$ for all $i, 1 \leq i \leq n$.

Some global constraints G admit a decomposition δ_k that *preserves* GAC. That is, given any instance c of G and any domain D on $X(c)$, given any subdomain $D' \subseteq D$, GAC on $(X(c), D', \{c\})$ prunes the same values as GAC on the network obtained from $\delta_k((X(c), D, \{c\}))$ by reducing $D(x_i)$ to $D'(x_i)$ for all $x_i \in X(c)$. $\texttt{atmost}_{p,v}$ is a global constraint that admits a decomposition preserving GAC (see Example 87). But there are some constraints, such as the `alldifferent`, for which we do not know any such decomposition.[12] For those constraints, it is sometimes possible to build a specialized algorithm that enforces GAC in polynomial time on all instances of the global constraint. For instance, Knuth and Raghunathan, and Régin, made the link between GAC on the `alldifferent` constraint and the problem of finding maximal matchings in a bipartite graph [77, 106], which is polynomial.

In [20], Bessiere et al. relaxed the definition of decomposition. They allow decompositions using constraints with unbounded arity as long as enforcing GAC on them is polynomial. The decomposition of a global constraint G is *GAC-polytime* if for any instance c of G and any domain on $X(c)$, enforcing GAC on the decomposition is polynomial. This enlarges the set of global constraints that can be decomposed. Nevertheless, there are global constraints for which we do not know any GAC-polytime decomposition that preserves GAC. Tools of computational complexity help us decide when a given global constraint has no chance to allow a GAC-polytime decomposition preserving GAC. In fact, if enforcing GAC on a global constraint G is NP-hard, there does not exist any GAC-polytime decomposition that preserves GAC (assuming P$\neq$ NP). For instance, enforcing GAC on `NValue` is NP-hard. This tells us that there is no way to find a GAC-polytime decomposition on which GAC always removes all GAC inconsistent values of the original `NValue` constraint.

Decompositions were limited to transformations in polynomial time, and so polynomial space. If we remove these restrictions, any global constraint allows a transformation into a binary network via the hidden variable encoding, where the unique additional variable has a domain of exponential size [45, 108]. GAC on this transformation is equivalent to GAC on the original constraint, even if

[11] Some decompositions depend only on the instance c of the global constraint and not on the domain. However, in other decompositions, the domain of the new variables depends on the domain of the variables in $X(c)$.

[12] In [26], Bessiere and Van Hentenryck characterized three types of globality for global constraints, depending on the non existence of decompositions preserving the solutions, preserving GAC or preserving the complexity of enforcing GAC.

enforcing GAC on it is NP-hard.

It is sometimes possible to express a global constraint as a combination of simpler constraints which is not a conjunction. Disjunctions are not naturally handled by constraint solvers. Van Hentenryck et al. proposed constructive disjunction as a way to partially propagate disjunctions of constraints [70]. Given a constraint $c = c_1 \vee c_2 \vee \ldots \vee c_k$, constructive disjunction propagates constraints c_i one by one independently of the others, and finally prunes values that were inconsistent with all c_i. This technique has been refined by Lhomme [84]. Bacchus and Walsh proposed a constraint algebra in which we can define meta-constraints as logical expressions composed of simpler constraints [4]. They give ways to propagate them and conditions under which GAC is guaranteed.

When enforcing GAC is too expensive on a global constraint, another possibility is to enforce a weaker level of consistency, such as BC(Z) or RC. BC(Z) and RC are significantly cheaper than GAC on constraints composed of arithmetic expressions (especially linear constraints). BC(Z) and RC are also used on other classes of constraints for which GAC is too expensive. In [80], Leconte showed that RC can be enforced on the `alldifferent` constraint at a cost asymptotically lower than that of Régin's GAC algorithm ([106]). Puget proposed a BC(Z) algorithm for `alldifferent` with an even lower complexity [103]. On the global cardinality constraint (`gcc`) defined by Régin [107], Quimper et al. showed that GAC is NP-hard if cardinalities are variables instead of fixed intervals [104]. Katriel and Thiel proposed a BC(Z) algorithm for `gcc` that runs in polynomial time even if cardinalities are variables [75]. In this case, BC(Z) is a means to propagate the constraint polynomially.

9.3 Creating propagators automatically

As an alternative to specialized algorithms for propagating a specific constraint, Apt and Montfroy proposed to generate sets of reduction rules [3]. A rule is of the form "if x_{i_1} takes value in S_{i_1}, ..., x_{i_k} takes value in S_{i_k} then y cannot take value v", where x_{i_j}'s and y belong to the scheme of the constraint and S_{i_j}'s are subsets of given domains $D(x_{i_j})$. For any constraint, there exists a set of rules that simulates arc consistency. However, its size can be exponential in the number of variables in the scheme of the constraint.

To avoid this combinatorial explosion, Dao et al. proposed to restrict their attention to rules in which variables x_{i_j} take values in intervals I_{i_j} instead of arbitrary subsets of the domains S_{i_j} [39]. This reduces the space of possibilities and permits to express the task of generating rules as a linear program to be solved by a simplex.

Another way to avoid combinatorial explosion when building propagators for a constraint c_{adhoc} is to take into account the internal structure of the constraint to factorize many satisfying tuples under the same rule. Barták proposed to decompose ad hoc binary constraints $c_{adhoc}(x_i, x_j)$ into *rectangles* [5]. The Cartesian product $r = S_i \times S_j$ of two sets of integers S_i and S_j is a rectangle for c_{adhoc} if $(v_i, v_j) \in S_i \times S_j \Rightarrow (v_i, v_j) \in c_{adhoc}$. Given a collection

R of rectangles such that $\bigcup_{r \in R} r = c_{adhoc}$, Barták gives a propagation algorithm that revises c_{adhoc} more efficiently than a generic algorithm. Cheng et al. extended this technique by proposing to decompose (possibly non-binary) constraints into 'triangles' instead of rectangles [30]. More precisely, they decompose a constraint $c_{adhoc}(x_1, \ldots, x_k)$ into a disjunction of 'box constraints'. A box is a k-dimensional hypercube $[l_1..u_1] \times \cdots \times [l_k..u_k]$ where $[l_i..u_i]$ is an interval of integers for x_i. A box constraint is the conjunction of a box B and a *simple* constraint c_b, that is, a constraint of the form $\sum_1^k a_i x_i \leq a_0$ (the set of allowed tuples looks like a triangle when the constraint is binary). Cheng et al. proposed an algorithm that generates a representation of the constraint c_{adhoc} as a disjunction of box constraints. Applying constructive disjunction on this representation is equivalent to arc consistency on c_{adhoc}.

9.4 Priorities in the propagation list

A simple way to improve the efficiency of propagation in constraint solvers is to put priorities on the different propagation events of the different constraints. We saw in Section 4.3 that the propagation list of arc consistency algorithms can be heuristically ordered. The main criterion in the case of generic AC algorithms for binary constraints was to put first the constraints that are expected to prune more. Constraint solvers contain various types of constraints and various types of propagation events for these constraints which can have different complexities. Laburthe et al. in [78] and Schulte and Stuckey in [111] proposed to maintain a propagation list with several levels of priority. The idea is to put a propagation event in a different level of the list depending on its time complexity. An event in the ith level is not popped while the $(i-1)$th level is not empty. The instantiation event on a simple arithmetic constraint is the kind of event that is put at the first level. Propagating GAC on an expensive global constraint is put at the last level. Propagating BC(Z) on the same constraint will be put in some intermediate level. The Choco solver uses a propagation list with 7 levels of priority [78].

Acknowledgements

I would like to thank especially Charlotte Truchet and Peter van Beek for their careful reading of this report and their many valuable comments. Thanks also to Eric Bourreau for having checked the section on specific constraints, to Peter Stuckey for his advice for choosing the names of the different consistencies on bounds, to Roland Yap for some pointers in the literature, and to Toby Walsh for interesting discussions on global constraints. Finally, I am very grateful to E.C. Freuder for the epigraph he kindly gave me for introducing this report.

References

[1] K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

[2] K.R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[3] K.R. Apt and E. Montfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Proceedings CP'99*, pages 58–72, Alexandria VA, 1999.

[4] F. Bacchus and T. Walsh. Propagating logical combinations of constraints. In *Proceedings IJCAI'05*, pages 35–40, Edinburgh, Scotland, 2005.

[5] R. Barták. A general relation constraint: An implementation. In *Proceedings of CP'00 Workshop on Techniques for Implementing Constraint Programming Systems (TRICS)*, pages 30–40, Singapore, 2000.

[6] R. Barták and R. Erben. A new algorithm for singleton arc consistency. In *Proceedings FLAIRS'04*, Miami Beach FL, 2004. AAAI Press.

[7] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, 1983.

[8] N. Beldiceanu. Pruning for the minimum constraint family and for the number of distinct values constraint family. In *Proceedings CP'01*, pages 211–224, Paphos, Cyprus, 2001.

[9] N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, Swedish Institute of Computer Science, Kista, Sweden, May 2005.

[10] F. Benhamou, D.A. McAllester, and P. Van Hentenryck. Clp(intervals) revisited. In *Proceedings of the International Symposium on Logic Programming (ILPS'94)*, pages 124–138, Ithaca, New York, 1994.

[11] F. Benhamou and W. Older. Appying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.

[12] H. Bennaceur and M.S. Affane. Partition-k-ac: an efficient filtering technique combining domain partition and arc consistency. In *Proceedings CP'01*, pages 560–564, Paphos, Cyprus, 2001. Short paper.

[13] P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings IEEE Conference on Artificial Intelligence and Applications (CAIA'95)*, 1995.

[14] C. Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

[15] C. Bessiere and M.O. Cordier. Arc-consistency and arc-consistency again. In *Proceedings AAAI'93*, pages 108–113, Washington D.C., 1993.

[16] C. Bessiere and R. Debruyne. Theoretical analysis of singleton arc consistency. In B. Hnich, editor, *Proceedings ECAI'04 Workshop on Modelling and solving problems with constraints*, pages 20–29, Valencia, Spain, 2004.

[17] C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings IJCAI'05*, pages 54–59, Edinburgh, Scotland, 2005.

[18] C. Bessiere, E. C. Freuder, and J. C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.

[19] C. Bessiere, E.C. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[20] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. The complexity of global constraints. In *Proceedings AAAI'04*, pages 112–117, San Jose CA, 2004.

[21] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. The tractability of global constraints. In *Proceedings CP'04*, pages 716–720, Toronto, Canada, 2004. Short paper.

[22] C. Bessiere, P. Meseguer, E.C. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.

[23] C. Bessiere and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.

[24] C. Bessiere and J.C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.

[25] C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165:165–185, 2005.

[26] C. Bessiere and P. Van Hentenryck. To be or not to be ... a global constraint. In *Proceedings CP'03*, pages 789–794, Kinsale, Ireland, 2003. Short paper.

[27] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst.*, 3(4):353–387, 1981.

[28] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *Proceedings of the CP'04 Workshop on Constraint Propagation and Implementation*, pages 29–43, Toronto, Canada, 2004.

[29] M. Cadoli and F.M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

[30] K.C.K. Cheng, J.H.M. Lee, and P.J. Stuckey. Box constraint collections for adhoc constraints. In *Proceedings CP'03*, pages 214–228, Kinsale, Ireland, 2003.

[31] A. Chmeiss and P. Jégou. Path-consistency: when space misses time. In *Proceedings AAAI'96*, pages 196–201, Portland OR, 1996.

[32] A. Chmeiss and P. Jégou. Sur la consistance de chemin et ses formes partielles. In *Proceedings RFIA'96*, pages 212–219, Rennes, France, 1996. (in French).

[33] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.

[34] A. Chmeiss and L. Saïs. About the use of local consistency in solving csps. In *Proceedings IEEE-ICTAI'00*, pages 104–107, Vancouver, Canada, 2000.

[35] C.W. Choi, W. Harvey, J.H.M. Lee, and P.J. Stuckey. Finite domain bounds consistency revisited. http://arxiv.org/abs/cs.AI/0412021, December 2004.

[36] H. Collavizza, F. Delobel, and M. Rueher. A note on partial consistencies over continuous domains. In *Proceedings CP'98*, pages 147–161, Pisa, Italy, 1998.

[37] M.C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989/90.

[38] H. Cros. *Compilation et apprentissage dans les réseaux de contraintes*. PhD thesis, University Montpellier II, France, 2003. in French.

[39] T.B.H. Dao, A. Lallouet, A. Legtchenko, and L. Martin. Indexical-based solver learning. In *Proceedings CP'02*, pages 541–555, Ithaca NY, 2002.

[40] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[41] R. Debruyne. A property of path inverse consistency leading to an optimal pic algorithm. In *Proceedings ECAI'00*, pages 88–92, Berlin, Germany, 2000.

[42] R. Debruyne and C. Bessiere. From restricted path consistency to max-restricted path consistency. In *Proceedings CP'97*, pages 312–326, Linz, Austria, 1997.

[43] R. Debruyne and C. Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings IJCAI'97*, pages 412–417, Nagoya, Japan, 1997.

[44] R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[45] R. Dechter. On the expressiveness of networks with hidden variables. In *Proceedings AAAI'90*, pages 556–562, Boston MA, 1990.

[46] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

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

[48] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[49] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

[50] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language chip. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.

[51] R.E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[52] J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia PA, 1995.

[53] E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, Nov 1978.

[54] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, Jan. 1982.

[55] E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, Oct. 1985.

[56] E.C. Freuder. Completable representations of constraint satisfaction problems. In *Proceedings KR'91*, pages 186–195, Cambridge MA, 1991.

[57] E.C. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings AAAI'96*, pages 202–208, Portland OR, 1996.

[58] E.C. Freuder and R.J. Wallace. Selective relaxation for constraint satisfaction problems. In *IEEE-ICTAI'91*, pages 332–339, San Jose CA, 1991.

[59] T.W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[60] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to NP-Completeness.* Freeman, San Francisco CA, 1979.

[61] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings Twelfth Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974.

[62] J. Gaschnig. Experimental case studies of backtrack vs waltz-type vs new algorithms for satisficing assignment problems. In *Proceedings CC-SCSI'78*, pages 268–277, 1978.

[63] I. Gent, K. Stergiou, and T. Walsh. Decomposable constraints. *Artificial Intelligence*, 123:133–156, 2000.

[64] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proceedings CP'97*, pages 327–340, Linz, Austria, 1997.

[65] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings AAAI'96*, pages 246–252, Portland OR, 1996.

[66] S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, October 1965.

[67] M. Gyssens. On the complexity of join dependencies. *ACM Trans. Database Syst.*, 11(1):81–108, 1986.

[68] C.C. Han and C.H. Lee. Comments on mohr and henderson's path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.

[69] R.M. Haralick and G.L. Elliott. Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[70] P. Van Hentenryck, V. Saraswat, and Y. Deville. The design, implementation, and evaluation of the constraint language `cc(FD)`. In *Constraint Programming: Basics and Trends.* Springer Verlag, 1995.

[71] ILOG. *User's manual.* ILOG Solver 4.4, ILOG S.A., 1999.

[72] P. Janssen, P. Jégou, B. Nouguier, and M. C. Vilarem. A filtering process for general constraint-satisfaction problems: Achieving pairewise-consistency using an associated binary representation. In *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, Fairfax VA, 1989.

[73] P. Jégou. *Contribution á l'étude des problèmes de satisfaction de contraintes: algorithmes de propagation et de résolution; propagation de contraintes dans les réseaux dynamiques.* PhD thesis, CRIM, University Montpellier II, 1991. in French.

[74] P. Jégou. On the consistency of general constraint-satisfaction problems. In *Proceedings AAAI'93*, pages 114–119, Washington D.C., 1993.

[75] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In *Proceedings CP'03*, pages 437–451, Kinsale, Ireland, 2003.

[76] D.E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[77] D.E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal of Discrete Mathematics*, 5(3):422–427, 1992.

[78] F. Laburthe and Ocre. Choco : implémentation du noyau d'un système de contraintes. In *Proceedings JNPC'00*, pages 151–165, Marseilles, France, 2000.

[79] J.L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[80] M. Leconte. A bounds-based reduction scheme for difference constraints. In *Proceedings of the FLAIRS'96 workshop on Constraint-based Reasoning (Constraint'96)*, Key West FL, 1996.

[81] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings CP'03*, pages 480–494, Kinsale, Ireland, 2003.

[82] C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings IJCAI'05*, pages 199–204, Edinburgh, Scotland, 2005.

[83] O. Lhomme. Consistency techniques for numeric csps. In *Proceedings IJCAI'93*, pages 232–238, Chambéry, France, 1993.

[84] O. Lhomme. Efficient filtering algorithm for disjunction of constraints. In *Proceedings CP'03*, pages 904–908, Kinsale, Ireland, 2003.

[85] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings IJCAI'97*, pages 366–371, Nagoya, Japan, 1997.

[86] A.K. Mackworth. Consistency in networks of relations. Technical Report 75-3, Dept. of Computer Science, Univ. of B.C. Vancouver, 1975. (also in Artificial Intelligence 8, 99-118, 1977).

[87] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[88] A.K. Mackworth. On reading sketch maps. In *Proceedings IJCAI'77*, pages 598–606, Cambridge MA, 1977.

[89] 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.

[90] P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings 5th International Conference on Integer Programming and Combinatorial Optimization (IPCO'96)*, volume 1084 of *LNCS*, pages 389–403, Vancouver, BC, 1996. Springer–Verlag.

[91] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphism. *Information Science*, 19:229–250, 1979.

[92] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[93] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings ECAI'88*, pages 651–656, Munchen, FRG, 1988.

[94] R. Mohr and G. Masini. Running efficiently arc consistency. In G. Ferraté et al., editor, *Syntactic and Structural Pattern Recognition*, pages 217–231. Springer–Verlag, Berlin, 1988.

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

[96] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[97] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings International Design Automation Conference (DAC-01)*, pages 530–535, Las Vegas NV, 2001.

[98] B.A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L.Kanal and V.Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[99] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[100] F. Pachet and P. Roy. Automatic generation of music programs. In *Proceedings CP'99*, pages 331–345, Alexandria VA, 1999.

[101] C.H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *J. Comput. System Sci.*, 28:244–259, 1984.

[102] P. Prosser, K. Stergiou, and T Walsh. Singleton consistencies. In *Proceedings CP'00*, pages 353–368, Singapore, 2000.

[103] J.F. Puget. A fast algorithm for the bound concictency of alldiff constraints. In *Proceedings AAAI'98*, pages 359–366, Madison WI, 1998.

[104] C.G. Quimper, A. López-Ortiz, P. van Beek, and A. Golynski. Improved algorithms for the global cardinality constraint. In *Proceedings CP'04*, pages 542–556, Toronto, Canada, 2004.

[105] C.G. Quimper, P. van Beek, A. López-Ortiz, A. Golynski, and S.B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings CP'03*, pages 600–614, Kinsale, Ireland, 2003.

[106] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings AAAI'94*, pages 362–367, Seattle WA, 1994.

[107] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings AAAI'96*, pages 209–215, Portland OR, 1996.

[108] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings ECAI'90*, pages 550–556, Stockholm, Sweden, 1990.

[109] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[110] C. Schulte and P.J. Stuckey. When do bounds and domain propagation lead to the same search space. In *Proceedings of the Third International Conference on Principles and Practice of Declarative Programming*, pages 115–126, Florence, Italy, September 2001. ACM Press.

[111] C. Schulte and P.J. Stuckey. Speeding up constraint propagation. In *Proceedings CP'04*, pages 619–633, Toronto, Canada, 2004.

[112] M. Singh. Path consistency revisited. *International Journal on Artificial Intelligence Tools*, 5(1-2):127–141, 1996.

[113] E. Tsang. No more 'partial' and 'full' looking ahead. *Artificial Intelligence*, 98:351–361, 1998.

[114] M.R.C. van Dongen. Lightweight arc-consistency algorithms. Technical Report TR-01-2003, Cork Constraint Computation Center, 2003.

[115] M.R.C. van Dongen and J.A. Bowen. Improving arc-consistency algorithms with double-support checks. In *Proceedings of the Eleventh Irish Conference on Artificial Intelligence and Cognitive Science*, pages 140–149, 2000.

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

[117] P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings ICLP'91*, pages 745–759, Paris, France, 1991.

[118] P. Van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

[119] P. Van Hentenryck, V.A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language `cc(FD)`. *Journal of Logic Programming*, 37(1-3):139–164, 1998.

[120] G. Verfaillie, D. Martinez, and C. Bessiere. A generic customizable framework for inverse local consistency. In *Proceedings AAAI'99*, pages 169–174, Orlando FL, 1999.

[121] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *Proceedings IJCAI'93*, pages 239–245, Chambéry, France, 1993.

[122] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.

[123] T. Walsh. Errata on singleton consistencies. Private communication, September 2000.

[124] T. Walsh. Relational consistencies. Technical Report APES report 28-2001, University of York, 2001.

[125] D.L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Tech.Rep. MAC AI-271, MIT, 1972.

[126] Y. Zhang and R.H.C. Yap. Arc consistency on n-ary monotonic and linear constraints. In *Proceedings CP'00*, pages 470–483, Singapore, 2000.

[127] Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings IJCAI'01*, pages 316–321, Seattle WA, 2001.