# Constraint Reasoning

Christian Bessiere

# Constraint Reasoning *

Christian Bessiere

**Abstract** In this chapter, I briefly present constraint reasoning. Constraint reasoning has been a subfield of artificial intelligence (AI) that is nowadays more well-known as constraint programming (CP). The change of name occurred more or less when CP has started to be widely used for solving combinatorial problems in industrial applications. This also corresponds to the moment where CP was enriched by the contributions from logic programming for the aspects related to languages and from operation research for the propagation of complex constraints. Considering the topic of this book, I will stay on a AI-oriented presentation of CP.

## 1 Introduction

The notion of constraint seen as a restriction of the combinations of values that a set of variables can take is sufficiently natural to have appeared early in the history of artificial intelligence. We can mention the work of Fikes [1970] on the REF-ART system, or the work of Waltz [1972] on the interpretation of contours. However, we usually associate the birth of constraint reasoning to the seminal paper of Montanari [1974], who formally defines for the first time what a constraint network is. This paper has been followed by other equally important papers by Mackworth [1977a] and Freuder [1978; 1982; 1985], who will lead research towards local consistencies and constraint propagation, notions which are specific to constraint reasoning. Laurière and his system ALICE [1978] can be seen as a pioneer of constraint-based solvers.

Christian Bessiere
CNRS, University of Montpellier, France,
e-mail: bessiere@lirmm.fr

* This chapter belongs to the book [Marquis et al., 2020]. It is essentially the English version of a chapter written in 2010 for the French book [Marquis et al., 2014]. A few references to more recent contributions have been added but the global structure has not been modified.

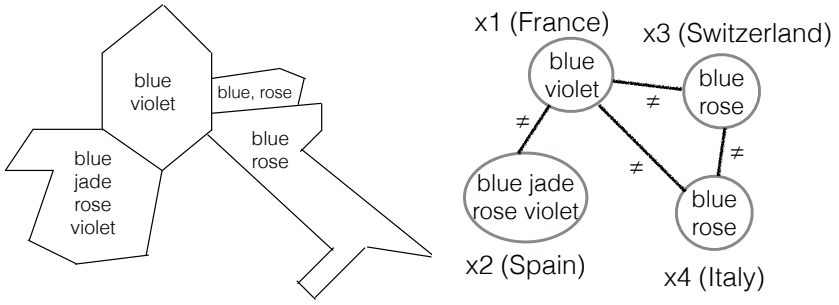


**Fig. 1** Map coloring problem where each country has only a subset of colors allowed (left). Constraint network representing the problem (right).

A constraint network consists of variables, each taking value in its respective domain, and constraints that restrict the possible combinations of values between variables. A constraint satisfaction problem (CSP) is the problem of deciding whether a given constraint network has solutions, that is, an assignment of values to all variables that satisfies all constraints. For instance, we can represent the map coloring problem shown in Figure 1 (left) by a constraint network where each country is represented by a variable whose domain is the set of colors available for that country (see Figure 1 (right)). Each pair of variables representing two neighboring countries is connected by a constraint specifying that the values taken by these two variables must be different. It is easy to be convinced that a solution to this constraint network corresponds to a satisfying coloring of our map. Constraint programming/reasoning covers all techniques related to solving CSPs.

A first advantage of CP, as already highlighted by Freuder in 1993 during a tutorial at IJCAI, is to provide a stable formalism that serves as input for solving algorithms. If you can represent your problem as a constraint network, whose solutions are representations of the solutions to your problem, you can use any constraint satisfaction algorithm to find your solution.

CP has been used in numerous applications. Among the first successes, we can cite optimization of the route of coal excavators in mines [ILOG, 1997], optimization of car assembly lines [Dincbas et al., 1988], search for structures in RNA sequences [Gaspin and Westhof, 1994], nurse rostering in hospitals [Cheng et al., 1997], radio link frequency assignment [Cabon et al., 1999], or assignment of platforms to trains [ILOG, 1997].

There exist other formalisms in which a combinatorial problem can be modeled and then solved. SAT or integer linear programming are such examples. One advantage of CP compared to these formalisms is its expressiveness. Many real applications contain *patterns*, such as the constraint 'all different', that requires a set of variables to take different values. These patterns can be encapsulated in a single constraint. They allow the user to easily model a whole piece of her problem, and they allow the algorithms to reason globally on these patterns, which otherwise

would have been broken into subparts. Reasoning from the original pattern allows to remain close to the problem definition and in some cases to infer more information than on any combination of subparts.

## 2 Definitions

In this section, I give the definitions needed to understand the rest of this chapter. I have as much as possible favored simplicity to rigor when there was no possible ambiguity on the interpretation.

**Definition 1 (Constraint Network).** A constraint network $N = (X, D, C)$ is composed of:

- a finite set $X = \{x_1, x_2, \ldots, x_n\}$ of integer *variables*,
- a *domain* on $X$, that is a set $D = D(x_1) \times \ldots \times D(x_n)$, where $D(x_i) \subset \mathbb{Z}$ is finite and given in extension, and
- a set $C = \{c_1, \ldots, c_e\}$ of *constraints*. A constraint $c_j \in C$ is a relation (or equivalently a Boolean function) defined on a sequence of variables $X(c_j) = (x_{j_1}, \ldots, x_{j_{|X(c_j)|}})$, called the *scope* of $c_j$. $c_j$ is a subset of $\mathbb{Z}^{|X(c_j)|}$.

The size of the network $N$ is often approximated through the parameters $n = |X|$, $d = max_{i \in 1..n}|D(x_i)|$, $e = |C|$ and $r = max_{j \in 1..e}|X(c_j)|$.

Values in $D(x_i)$ could be of any type, but it simplifies the presentation to consider them as integers. It is not a restriction as $D(x_i)$ is finite. An element of $\mathbb{Z}^{|X(c_j)|}$ is called a *tuple*. A tuple in $\mathbb{Z}^{|X(c_j)|}$ is *valid* if and only if it belongs to $D^{X(c_j)} = D(x_{j_1}) \times \ldots \times D(x_{j_{|X(c_j)|}})$. Tuples belonging to $c_j$ *satisfy* $c_j$, the other tuples of $\mathbb{Z}^{|X(c_j)|}$ *violate* $c_j$.

**Example 1** The map coloring example of Figure 1 (left) can be represented by the constraint network of Figure 1 (right). Variables $x_1, \ldots, x_4$ respectively correspond to the countries France, Spain, Switzerland, Italy. The domains are $D(x_1) = \{B, V\}, D(x_2) = \{B, J, R, V\}$, $D(x_3) = D(x_4) = \{B, R\}$, where $B, J, R, V$ are integers representing colors blue, jade, rose, violet. The constraint between France and Spain, say $c_1$, which ensures that France (variable $x_1$) and Spain (variable $x_2$) take different colors, can be defined by $X(c_1) = (x_1, x_2)$ and $c_1 = \{(B,J), (B,R), (B,V), (V,B), (V,J), (V,R)\}$, or simply by $x_1 \neq x_2$. ♢

The notion of instantiation is central to the rest of this chapter.

**Definition 2 (Instantiation).** An *instantiation $I$ on* a set $Y \subseteq X$ of variables is an assignment of a value of domain $D(x_i)$ to each variable $x_i$ in $Y$.[2] If $W$ is a subset of $Y$, we denote by $I[W]$ the restriction (or projection) of $I$ to variables in $W$. An

[2] In the rest of this chapter we will represent an instantiation indifferently as a sequence or as a set of variable assignments. The assignment of value $v_i$ to variable $x_i$ is denoted by $(x_i, v_i)$.

instantiation *satisfies* a constraint $c_j$ if $X(c_j) \subseteq Y$ and $I[X(c_j)] \in c_j$. If $X(c_j) \subseteq Y$ and $I[X(c_j)] \notin c_j$, $I$ *violates* $c_j$. An instantiation is *locally consistent* if and only if none of the constraints in $C$ is violated.

**Example 2** On our map coloring example, $\{(x_1,B);(x_2,J);(x_4,B)\}$ is an instantiation on $\{x_1,x_2,x_4\}$ (i.e., on France, Spain and Italy) that is not locally consistent because it violates the constraint on $x_1$ and $x_4$. $\{(x_1,B);(x_2,J);(x_4,R)\}$ is a locally consistent instantiation. ♢

**Definition 3 (Solution).** A *solution* $S$ of a constraint network $N = (X,D,C)$ is an instantiation on $X$ that does not violate any constraint of $C$. We denote by $Sol(N)$ the set of the solutions of $N$.

**Definition 4 (CSP).** The *constraint satisfaction problem (CSP)* is defined by:

Instance : A constraint network $N = (X,D,C)$
Question : $Sol(N) \neq \emptyset$ ?

**Theorem 1.** *CSP is NP-complete.*

*Proof.* Direct consequence of the fact that SAT, the first problem shown NP-complete, is a particular CSP where variables are Boolean and constraints are clauses. □

I have described the formalism and the central question. In the next section I will present the basic techniques for tackling this question.

## 3 Chronological Backtracking

Solving a CSP can be done by generating all possible instantiations on $X$ until we find one which is solution to the constraint network. This technique, called "generate and test" is a bit too brute force to deserve being presented. I will directly present the slightly improved version called *chronological backtracking (BT)* [Golomb and Baumert, 1965].

The function BT, presented in Algorithm 1, takes as input the constraint network $N$ and a locally consistent instantiation $I$. The call $BT(N,\emptyset)$ prints the first solution found and returns **true** if $N$ has solutions, **false** otherwise. BT first checks whether $I$ is a complete instantiation (line 1). If yes, this means that $I$ is a solution. BT prints it and returns **true**. If $I$ is not a complete instantiation, BT selects a not yet instantiated variable $x_i$ (line 2) and tries its values one by one until it finds one that can be extended into a solution. Once a value $v_i$ is chosen, if the instantiation $I$ extended by the assignment $x_i \leftarrow v_i$ is locally consistent (line 4), the function BT is recursively called with the new instantiation (line 5). If all values of $D(x_i)$ have been tried without leading to a solution, BT returns **false** (line 6). We call this step a *backtrack*. If a solution is found by BT without any backtrack, we say that the order

**Algorithme 1 :** Chronological backtracking

```
function BT (N: network; I: instantiation): Boolean
  begin
1   if |I| = |X| then print I; return true
2   choose a variable x_i not yet instantiated
3   foreach v_i ∈ D(x_i) do
4       if I ∪ {(x_i, v_i)} is locally consistent then
5           if BT(N, I ∪ {(x_i, v_i)}) then return true
6   return false
```

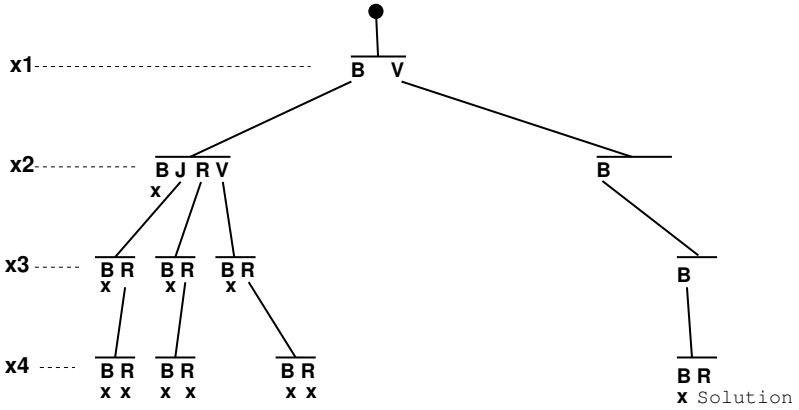


**Fig. 2** Search tree of BT on the constraint network of Example 1.

used is *backtrack-free* for this network. The space explored by BT is called the *search tree*. The instantiations are the nodes. They have an incoming edge coming from the preceding locally consistent instantiation. The root of the tree is the empty instantiation. Figure 2 displays the search tree explored by BT when called on the network of Example 1, with variables and values selected in lexicographic order (lines 2 and 3).

## 4 Constraint Propagation

There are several ways to reduce the size of the search tree explored by the function BT. Constraint propagation is the most important one. Constraint propagation consists in explicitly forbidding values or combinations of values for variables that cannot be extended to any solution of the constraint network. For instance, given a network involving two variables $x_1$ and $x_2$ with domains 1..10, and a constraint specifying that $|x_1 - x_2| > 5$, propagating the constraint allows us to forbid values 5

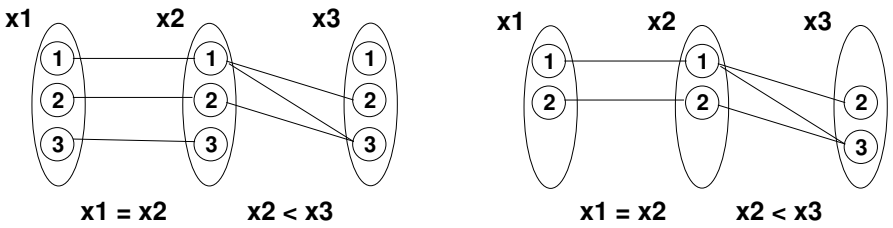


**Fig. 3** Constraint network of Example 3 before arc consistency (left) and after (right). Edges represent the pairs of values allowed by a constraint.

and 6 for $x_1$ and $x_2$. By removing these values from the domains of $x_1$ and $x_2$, we reduce the size of the search tree that the function BT has to explore.

Constraint propagation can be presented from two points of view: *local consistencies* and *iteration of reduction rules*. Local consistencies define a property that the network must satisfy *after* the propagation phase. The operational behavior is not specified. On the contrary, reduction rules describe the propagation mechanism itself. Rules are conditions on the type of operations of reduction that can be applied on the network. I will present constraint propagation essentially through local consistencies.

## *4.1 Consistency on one constraint at a time*

### Arc consistency

Arc consistency is the most well-known local consistency. Arc consistency ensures that for every constraint in the network, for every variable in its scope, each value in the domain of the variable belongs to a valid tuple satisfying the constraint. Arc consistency has been clearly defined for the first time in the seminal paper of Mackworth [1977a]. (Even if the idea of arc consistency has been used before, e.g., [Waltz, 1972].)

**Example 3** Let $N$ be a network involving three variables $x_1, x_2$ and $x_3$ with domains $D(x_1) = D(x_2) = D(x_3) = \{1,2,3\}$, and the constraints $x_1 = x_2$ and $x_2 < x_3$ (see Figure 3). $N$ is not arc consistent because there are values that are not compatible with some constraints. On constraint $x_2 < x_3$ we observe that value 3 for $x_2$ must be deleted because there is no value greater than 3 in $D(x_3)$. We can also delete value 1 from $D(x_3)$ because of constraint $x_2 < x_3$. The deletion of value 3 from $D(x_2)$ leads to the deletion of value 3 from $D(x_1)$ because of $x_1 = x_2$. All values of all variables are now compatible with all constraints. ♢

I give the definition of arc consistency in its more general form, that is, for constraints of any arity. When arity is greater than 2, arc consistency is often called

generalized arc consistency —and also sometimes hyper arc consistency in old papers.

**Definition 5 (Arc consistency (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 $D$ if and only if there exists a valid tuple $\tau$ satisfying $c$ and $v_i = \tau[\{x_i\}]$. Such a tuple is called a *support* for $(x_i, v_i)$ on $c$.
- The constraint $c$ is arc consistent on $D$ if and only if all values of all variables in $X(c)$ have a support on $c$.
- The network $N$ is *arc consistent* if and only if all constraints in $C$ are arc consistent on $D$.

Enforcing arc consistency on the network $N = (X, D, C)$ is done by computing the *arc consistency closure* $AC(N)$ of $N$. $AC(N)$ is the network $(X, D_{AC}, C)$, where $D_{AC} = \cup\{D' \subseteq D \mid (X, D', C) \text{ is arc consistent}\}$. $AC(N)$ is arc consistent and is *unique*. It has the same solutions as $N$. $AC(N)$ is computed by iteratively removing the values that do not have a support on a constraint until reaching the fixed point where all values have support on all constraints.

We will see later that enforcing arc consistency on a constraint network is an essential task in the search for solutions. Proposing efficient arc consistency algorithms has thus always been a central topic of research in the community. The new ideas proposed to improve arc consistency algorithms are often used to improve propagation algorithms for other kinds of local consistency that will be discussed later.

AC3 is the most well-known and simplest algorithm for enforcing arc consistency. It has been proposed in [Mackworth, 1977a] for binary constraints. Algorithm 2 describes it in its form for constraints of any arity (as proposed in [Mackworth, 1977b]).

The main component of AC3 is the revision of an arc, that is, the removal of the values of a variable that are not consistent on a given constraint. The name "arc" comes from the binary case. The function `Revise`$(x_i, c)$ processes each value $v_i$ from $D(x_i)$ (line 2), and explores $D^{X(c)\setminus\{x_i\}}$ to find 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 modified is memorized in the Boolean CHANGE (lines 4–5). The function returns true if the domain $D(x_i)$ has been modified, false otherwise (line 6).

The AC3 algorithm is composed of a simple loop revising arcs until no more value removals occur. The domains are then all arc consistent on all constraints. To avoid too many useless calls to `Revise`, AC3 maintains a list $Q$ of all pairs $(x_i, c)$ for which we are not sure that $D(x_i)$ is arc consistent on $c$. At line 7, $Q$ is initialized with all pairs $(x_i, c)$ such that $x_i \in X(c)$. The main loop (line 8) picks pairs $(x_i, c)$ one by one in $Q$ (line 9) and calls `Revise`$(x_i, c)$ (line 10). If $D(x_i)$ becomes empty, AC3 returns false (line 11). Otherwise, if $D(x_i)$ has been changed, it is possible that a value from another variable $x_j$ has lost its supports on a constraint $c'$ involving both $x_i$ and $x_j$. Thus, all pairs $(x_j, c')$ such that $x_i, x_j \in X(c')$ must be put again in $Q$

**Algorithme 2 :** AC3 (also denoted by GAC3 on non-binary constraints)

```
function Revise(in x_i: variable; c: constraint): Boolean
  begin
1   |  CHANGE ← false
2   |  foreach v_i ∈ D(x_i) do
3   |  |  if ∄ a valid tuple τ ∈ c with τ[x_i] = v_i then
4   |  |  |  remove v_i from D(x_i)
5   |  |  |  CHANGE ← true
6   |  return CHANGE

function AC3(in X: set): Boolean
  begin
7   |  Q ← {(x_i, c) | c ∈ C, x_i ∈ X(c)}
8   |  while Q ≠ ∅ do
9   |  |  pick (x_i, c) from Q
10  |  |  if Revise(x_i, c) then
11  |  |  |  if D(x_i) = ∅ then return false
12  |  |  |  else Q ← Q ∪ {(x_j, c') | c' ∈ C ∧ c' ≠ c ∧ j ≠ i ∧ x_i, x_j ∈ X(c')}
13  |  return true
```

(line 12). When $Q$ is empty, AC3 returns true (line 13) because all arcs have been revised and all remaining values of all variables are consistent with all constraints.

AC3 is polynomial in the arity of the constraints It is in $O(er^3d^{r+1})$, where $r$ is the greatest arity of constraints in $C$. On networks of binary constraints it gives us $O(ed^3)$. The complexity of AC3 is not optimal because `Revise` does not store any information from the computation of supports and must then do and redo the same tests of constraint at each call.

Many algorithms have been proposed to improve the complexity of AC3. Mohr et al. [1986; 1988] proposed AC4, the first optimal algorithm for arc consistency ($O(ed^2)$ on binary constraints, and $O(erd^r)$ in general), at the cost of an expensive data structure. Bessiere et al. [2001; 2005] proposed AC2001. It has the advantage to be based on the same simple schema as AC3 whereas having an optimal complexity thanks to a light structure of pointers. More recently, Ullmann [2007] has proposed STR, an algorithm for enforcing arc consistency on constraints represented by a table of satisfying tuples. STR uses a data structure of sparse sets that efficiently maintains and restores the set of valid tuples in the table during search. Lecoutre et al. have improved this technique in STR2 [2011] and STR3 [2015]. These improvements for arc consistency on table constraints culminate with the bitwise algorithm CT (for *Compact Table*) proposed by Demeulenaere et al. [2016].

### Bounds consistency

When a constraint involves many variables or when the domains of the variables are large, arc consistency can be too costly. An alternative is to apply *bounds consistency (BC)*, a local consistency weaker than arc consistency that uses the fact that domains are composed of integers, and thus inherit the total ordering on $\mathbb{Z}$. We define the smallest and greatest values in a domain $D(x_i)$, denoted by $min_D(x_i)$ and $max_D(x_i)$ respectively, and called the *bounds* of $D(x_i)$.

**Definition 6 (Bounds consistency).** Given a network $N = (X, D, C)$ and a constraint $c$, a *bound support* $\tau$ on $c$ is a tuple satisfying $c$ such that for all $x_i \in X(c), min_D(x_i) \leq \tau[x_i] \leq max_D(x_i)$. A constraint $c$ is *bounds consistent* if and only if 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 $c$. $N$ is bounds consistent if and only if all its constraints are bounds consistent.

**Example 4** Suppose we have the variables $x, y, z$, the domains $D(x) = \{0,5\}$, $D(y) = \{0,2,5,12\}$, $D(z) = \{4,5\}$, and the constraint $|x-y| = z$. BC only removes value 12 from $D(y)$ because it is the only bound without a bound support. Value 2 for $y$ does not have any bound support but it is not a bound, and value 4 for $z$ is a bound but it has bound supports, such as $((x,5),(y,1),(z,4))$ for instance. AC would have removed 2 and 12 for $y$ and 4 for $z$. ♢

### Domain reduction rules

A domain reduction rule specifies a sufficient condition for removing values. A propagation algorithm for reduction rules iterates on the rules until no more domain is modified, that is, a fixed point has been reached [Apt, 2003]. Domain reduction rules are useful when we have information on the semantics of the constraint to propagate. This allows us to specify simple and efficient ways to propagate the constraint. It is indeed often inefficient to use a generic algorithm such as AC3 when we have specific information on the constraint. This approach of domain reduction rules is used a lot in solvers because solvers usually contain many predefined basic constraints, especially arithmetic constraints. On these constraints, a change in a domain may have a different effect on the other variables of the constraint depending on whether a removed value is in the middle of the domain, is the minimum value, the maximum value, or the domain becomes a singleton. Solvers are thus often able to differentiate these types of changes in a domain, called *events*. Events recognized by the majority of solvers are:

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

Thanks to these four types of events we can specify domain reduction rules for many basic constraints. The approach of reduction rules differs from the approach

of local consistencies in the sense that it has an operational definition of its fixed point. A set of reduction rules reaches its own fixed point, which is not necessarily an existing level of local consistency such as bounds consistency or arc consistency. On the contrary, arc consistency is defined as a property independently of the way to achieve it.

**Example 5** Consider the constraint $\texttt{alldifferent}(x,y,z)$. Instead of calling the function `Revise` to propagate this constraint, which would be in $O(d^3)$, we can build the following set of rules:

R1: **if** $\texttt{Instantiate}(x)$ **then** $D(y) \leftarrow D(y) \setminus D(x)$; $D(z) \leftarrow D(z) \setminus D(x)$
R2: **if** $\texttt{Instantiate}(y)$ **then** $D(x) \leftarrow D(x) \setminus D(y)$; $D(z) \leftarrow D(z) \setminus D(y)$
R3: **if** $\texttt{Instantiate}(z)$ **then** $D(x) \leftarrow D(x) \setminus D(z)$; $D(y) \leftarrow D(y) \setminus D(z)$

Applying these rules is very efficient because each rule executes in constant time. The different types of events allow us to trigger a rule on a constraint only if it has chances to lead to other value removals. Suppose the domains are $D(x) = D(y) = D(z) = \{1,2,3,4\}$. If 1 and 2 are removed from $D(z)$, then $\texttt{Instantiate}(z)$ is false and none of the rules of the constraint $\texttt{alldifferent}(x,y,z)$ need to be triggered. If 1,2 and 4 are removed from $D(z)$, $\texttt{Instantiate}(z)$ is true and rule R3 is triggered.

The set of rules R1, R2, R3, is not sufficient to ensure arc consistency. Suppose 3 and 4 are removed from $D(x)$ et $D(y)$ while $D(z) = \{1,2,3,4\}$. Rules R1, R2, R3 do not remove any value whereas 1 and 2 in $D(z)$ are not arc consistent. ♢

## *4.2 Strong consistencies*

Arc consistency is not the only way to detect inconsistencies in a constraint network. Since the '70s, other properties have been proposed to discover more inconsistencies than arc consistency. Freuder [1978] proposed $k$-consistencies.

**Definition 7 ($k$-consistency).** Given a network $N = (X,D,C)$ and a set of variables $Y \subseteq X$ with $|Y| = k-1$, a locally consistent instantiation $I$ on $Y$ is $k$-consistent if and only if for any $k$th 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* if and only if for any set $Y$ of $k-1$ variables, any locally consistent instantiation on $Y$ is $k$-consistent.

Before Freuder proposed $k$-consistency, Montanari [1974] had proposed path consistency. Path consistency was defined for networks of binary constraints in which each pair of variables is involved in at most one constraint. On such networks, path consistency is equivalent to 3-consistency.

**Example 6** 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/3-consistent because neither $((x_1,1),(x_3,2))$ nor $((x_1,2),(x_3,1))$ can be extended to a value in the domain of $x_2$ satisfying both $c_{12}$ and $c_{23}$. The network $N' = (X,D,C \cup \{x_1 = x_3\})$ is path/3-consistent. ♢

Freuder [1982] has also defined strong $k$-consistency.

**Definition 8 (Strong $k$-consistency).** A network is *strongly $k$-consistent* if and only if it is $j$-consistent for all $j \leq k$.

When $k$ is large, enforcing $k$-consistency is far too expensive in practice, both in time and space. However, the notion of $k$-consistency is not useless because it allows us to give an operational understanding of the important notion of *global consistency*. A network is globally consistent when *all* locally consistent instantiations can be extended to solutions. It means that the function BT described in Algorithm 1 finds a solution or proves that none exists without any backtrack. A network is globally consistent if and only if it is strongly $|X|$-consistent.

Freuder [1985] has also proposed $(i, j)$-consistencies. $(i, j)$-consistency ensures that any locally consistent instantiation of size $i$ can be extended to any $j$ additional variables. $k$-consistency is $(k-1,1)$-consistency.

Janssen et al. [1989] proposed the first local consistency based on the number of constraints involved in the instantiations instead of the number of variables.

**Definition 9 (Pairwise consistency).** Given a network $N = (X,D,C)$, the pair of constraints $c_1$ and $c_2$ in $C$ is pairwise consistent if and only if any instantiation on $X(c_1)$ (resp. $X(c_2)$) satisfying $c_1$ (resp. $c_2$) can be extended to an instantiation on $X(c_1) \cup X(c_2)$ satisfying $c_2$ (resp. $c_1$). $N$ is *pairwise consistent* if and only if any pair of constraints in $C$ is pairwise consistent.

**Example 7** 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 arc consistent. However, it is not pairewise consistent because the tuple $(121)$ of $c_1$ is not compatible with any tuple in $c_2$. ♢

Other authors have proposed local consistencies that are parameterized by the number of constraints involved in the reasoning. Jégou [1993] proposed hyper $k$-consistency, a kind of dual of $k$-consistency based on constraints. Dechter and van Beek [1997] proposed $(i,m)$-*relational* consistency, a consistency that focuses on inconsistencies inside the scopes of existing constraints.

Local consistencies cited above have in common a major drawback for an easy integration in constraint solvers: they add new constraints that were not present in the original network and/or they modify existing constraints. (See Examples 6 and 7). These added/modified constraints are expensive to store and propagate because they do not have any particular semantics.

In the '90s, *domain filtering* consistencies were introduced. Domain filtering consistencies can remove more values from the domains than arc consistency but they keep the constraints unchanged [Berlandier, 1995; Freuder and Elfe, 1996; Debruyne and Bessiere, 1997; Bennaceur and Affane, 2001]. I will just present *singleton arc consistency (SAC)* [Debruyne and Bessiere, 2001]. SAC is the most well-known of these domain filtering consistencies because it has a very simple definition and it can easily be integrated in the architecture of the current generation of

solvers. SAC is based on the idea of refuting the selection of a value for a variable by arc consistency propagation.

**Definition 10 (Singleton arc consistency).** A network $N = (X,D,C)$ is *singleton arc consistent* if and only if for all $x_i \in X$, for all all $v_i \in D(x_i)$, the arc consistency closure of the subnetwork $N|_{x_i=v_i}$ does not have any empty domain. $N|_{x_i=v_i}$ is the network where the domain of $x_i$ in $N$ has been reduced to the singleton $\{v_i\}$,

An approach of the same vein as SAC had been proposed in numerical constraint networks under the name 3B-consistency [Lhomme, 1993], and in scheduling under the name *shaving* [Martin and Shmoys, 1996]. The difference with SAC is that only the bounds can be selected, and bounds consistency is used instead of arc consistency to refute the selected bound.

## 5 Polynomial Cases

As the CSP is NP-complete, it is natural to ask the question of the existence of particular classes of constraint networks on which the CSP is polynomial.

The first results in this direction have been proposed by Freuder. They are based on the *structure* of the constraint network. The structure of a network is represented either by its *primal graph* or by its *associated hypergraph*. The primal graph of a constraint network $N = (X,D,C)$ is the graph $G_N = (X,E)$ which has a node per variable in $N$ and an edge $\{x_i,x_j\}$ in $E$ if and only if there exists a constraint $c \in C$ with $\{x_i,x_j\} \subseteq X(c)$. The associated hypergraph of a constraint network $N = (X,D,C)$ is the hypergraph $H_N = (X,E)$ which has a node per variable in $N$ and an hyperedge $e$ in $E$ if and only if there exists a constraint $c \in C$ with $e = X(c)$. Freuder [1982] defines the *width* of the primal graph of a constraint network. The width is the smallest integer $k$ such that a greedy procedure removing a node each time it has a degree lower than or equal to $k$ will remove all nodes of the graph.[3] The reverse of the order in which nodes have been removed by the greedy procedure is an order of width $k$.

**Theorem 2 ([Freuder, 1982]).** *If a constraint network $N$ has a primal graph of width at most $k$ and is strongly $(k+1)$-consistent, then calling BT on a variable ordering of width at most $k$ is backtrack-free.*

In the same paper, Freuder underlines the limitations of this theorem. Enforcing strong $(k+1)$-consistency with $k > 1$ creates new constraints. These new constraints increase the width of the primal graph, making false the precondition of the theorem. Freuder characterizes the case where the structure is not modified by enforcing strong $k$-consistency.

[3] The width of Freuder is a lower bound to the *tree-width* [Arnborg, 1985].

**Corollary 1 ([Freuder, 1982, 1985]).** *If a constraint network has a primal graph of width* 1 *(tree/forest), it can be solved in polynomial time by enforcing arc consistency and then calling BT, which will be backtrack-free on any ordering of width* 1.

Corollary 1 only applies to networks of binary constraints because a ternary constraint induces a triangle in the primal graph. Janssen and Vilarem [1988] extended Corollary 1 to networks of non-binary constraints. They proved that if a constraint network has a Berge-acyclic associated hypergraph then arc consistency decides its satisfiability. (In other words, arc consistency detects a wipe out if and only if the network has no solution.) As a result, MAC finds a solution backtrack-free, that is, in polynomial time if all constraints can be made arc consistent in polynomial time. Cohen and Jeavons [2017] recently showed that the condition is tight because Berge-acyclicity is the only structural property that allows arc consistency to decide satisfiability. Berge-acyclicity is thus the only structural property that allows MAC to find a solution in polynomial time.

Many contributions have extended the results of Freuder to constraint networks with a structure more general than trees. Many of them lie on the property that if the *tree-width* of the primal graph is bounded, then the network is polynomial to solve [Freuder, 1990]. Other contributions directly use properties of the associated hypergraph [Gyssens et al., 1994; Gottlob et al., 2000].

There is an orthogonal approach to characterize polynomial classes. Instead of restricting the structure of the constraint network, we can restrict the type of relations of the constraints used in the network. Let us denote by $CSP(\Gamma)$ the restriction of CSP to networks in which the constraints are defined by relations from the language $\Gamma$. Cooper et al. [1994] have proposed the *zero-one-all (ZOA)* relations. A binary relation $c$ with $X(c) = (x_i, x_j)$ is ZOA if and only if each value $v_i$ for $x_i$ is compatible with either zero, or one, or all values of $x_j$.

**Theorem 3 ([Cooper et al., 1994]).** *If all constraints in a binary network are ZOA, then enforcing path consistency is sufficient to ensure that BT is backtrack-free on the resulting network, that is $CSP(ZOA)$ is polynomial.*

ZOA is one of the first examples of tractable language of relations. There exist many other languages of relations such that networks composed of constraints defined by these relations are polynomial to solve. Another early example is *connected row convex* relations [van Beek and Dechter, 1995; Deville et al., 1999]. Polymorphisms have been used to better understand the properties of languages. A relation $c$ is closed for a polymorphism $f$ of arity $k$ if for any set of $k$ tuples in $c$, the tuple obtained by applying $f$ component-wise produces a tuple in $c$. For instance, in order to be max-closed (i.e., closed for function $f = max$), a relation $c$ containing the tuples $(2,1,2,3)$ and $(1,2,1,1)$ must also contain the tuple $(2,2,2,3)$. Jeavons and Cooper [1995] have shown that $CSP(\Gamma)$ is polynomial if $\Gamma$ is the language of max-closed relations. More general properties of polymorphisms leading to tractability have been extensively studied. Jeavons et al. [1998] have linked near-unanimity polymorphisms to the level of $k$-consistency sufficient to ensure global consistency. Bulatov and Dalmau [2006] have shown that if $f$ is a *Mal'tsev* polymorphism then $CSP(\Gamma)$ is polynomial for any language of relations closed for $f$. Chen

et al. [2013] extended this result to *majority* polymorphisms. Feder and Vardi [1998] conjectured that for any finite constraint language $\Gamma$, $CSP(\Gamma)$ is either polynomial or NP-complete. This fundamental dichotomy conjecture has recently been proved independently by Bulatov [2017] and Zhuk [2017]. It has also recently been proved that given a language $\Gamma$, we can recognize in polynomial time whether $CSP(\Gamma)$ is polynomial or NP-complete [Carbonnel, 2016].

## 6 Solution Synthesis and Decompositions

Dechter and Pearl [1988] have proposed *Adaptive consistency (AdC)*. AdC uses Theorem 2 while taking into account the limitations raised by Freuder (see above). Instead of applying uniformly $k$-consistency everywhere in the network, AdC applies a limited form of consistency, unidirectional, and adapted to the number of neighbors preceding the processed variable. Let $<_o$ be a total ordering on the variables. $parents(x_i)$ denotes the set of parents of $x_i$ in the primal graph $G_N$. Variables are made AdC one by one from the last to the first according to $<_o$. A variable $x_i$ is made AdC by creating a constraint of scope $parents(x_i)$ which forbids any locally consistent instantiation of $parents(x_i)$ that cannot be extended to a locally consistent instantiation on $parents(x_i) \cup \{x_i\}$. All possible edges in $parents(x_i)$ are then added to $G_N$. If none of the constraints created by AdC is the empty constraint, the function BT applied according to the ordering $<_o$ finds a solution in a backtrack-free way. The complexity of AdC is $O(nd^{w+1})$, where $w$ is the induced width of the $<_o$ ordering, that is, the width of $<_o$ after having applied AdC. AdC requires a given variable ordering before the local consistency step. If the order is bad (i.e., it has a large induced width), AdC is expensive. With the best order, AdC is polynomial in the tree-width of the original primal graph. AdC has a dynamic programming flavor, and as such, it has strong similarities with results presented in [Bertelé and Brioshi, 1972].

Dechter and Pearl [1989] proposed *tree clustering*, a decomposition technique that transforms any network into a tree-structured network. Variables are clustered in baskets. Each basket is considered as a meta-variable. Two baskets containing variables involved in the same constraint are linked by a meta-constraint. Baskets are built in such a way that the network of meta-variables has the structure of a tree. All the local solutions on a basket are generated and become the values in the domain of the meta-variable representing this basket. As opposed to AdC, no variable ordering is required. However, generating all solutions of a basket can be very space consuming, especially when large baskets are required. Many other decomposition techniques have been proposed [Gottlob et al., 2000; Jégou and Terrioux, 2003].

We can also use the structure of the solutions instead of the structure of the network. Amilhastre et al. [2002] associate each constraint with a finite state automaton and combine these automata to obtain an automaton representing all solutions of the network. If the solutions are not too scattered, the automaton can be of reasonable

size, and many kind of queries (e.g., counting solutions), usually expensive (exponential time), become linear or constant time.

Finite state automata have also been used in the form of multivalued decision diagrams (MDDs) to store the domains of the variables [Andersen et al., 2007]. Storing the domains as an MDD allows us to capture the space of possible assignments for variables more precisely than the brute-force Cartesian product of the individual domains of variables. However, with an MDD, the way constraints are propagated differs from the standard case of individual domains. For each type of constraint, a new propagation algorithm has to be written as a combination of MDD operations [Hoda et al., 2010; Bergman et al., 2014].

## 7 Improving Chronological Backtracking

Solution synthesis and decomposition techniques are only practicable in the particular case where the network has the right structure (for instance close to a tree for tree clustering). The standard technique to solve CSPs is thus function BT. Many researchers have proposed techniques to make BT less inefficient. Adapting from Dechter and Pearl [1988], we can classify the types of improvements of BT in different categories that can be combined. Look back techniques use implicit information contained in a failure to avoid latter inconsistent branches in the search tree. Look ahead techniques apply some level of constraint propagation at each node of the search tree to eliminate as early as possible inconsistent branches. Finally, variable and value ordering heuristics change the order in which variables and values are instantiated to decrease the size of the search space.

### *7.1 Look back*

The most well-known look back technique is *backjumping*. There are several versions of backjumping. The first one was proposed by Gaschnig [1978; 1979], then improved by Dechter [1990], then by Prosser [1993]. The idea in backjumping is that when all the values for a variable $x_i$ have led to a failure, instead of backtracking to the previous variable in the order, we try to jump back as high as possible in the search tree while ensuring that we do not lose solutions. The way to do this is to jump back to the lowest variable involved in the failure of the values of $x_i$. Figure 4 presents the search tree explored by *confilct-directed backjumping (CBJ)*, the backjumping of Prosser, when called on the network of Example 1, if we select the variables and values in lexicographic order. We observe that CBJ behaves exactly like BT (see Figure 2) until we meet a variable whose values are all inconsistent with the current branch, that is, $x_4$. For each value $v$ for $x_4$, CBJ looks for the highest variable assignment that caused the inconsistency of $v$: $x_1 = B$ for value $B$ and $x_3 = R$ for $R$. CBJ then backtracks to the lowest variable ($x_3$) and stores $x_1$ as another possible

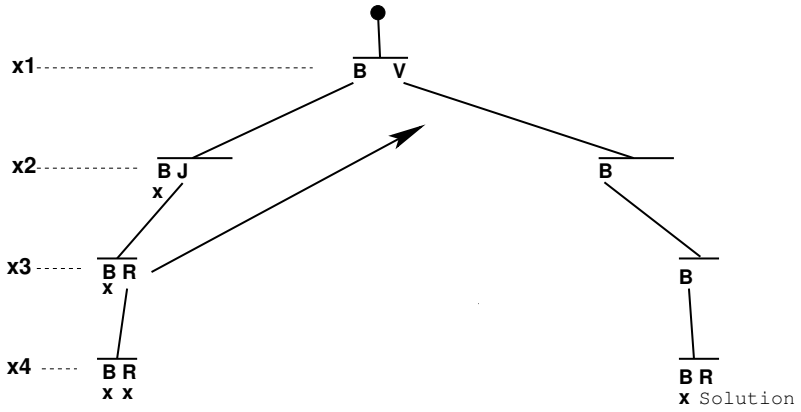


**Fig. 4** Search tree of CBJ on the constraint network of Example 1.

cause of failure. As $x_3$ has already tried all its values, it cannot change value and then applies the same process: Its value $B$ is inconsistent because of $x_1$, and according to $x_4$, its value $R$ cannot lead to a solution because of $x_1$. Thus, $x_3$ backjumps to the lowest possible culprit, that is, $x_1$. $x_1$ changes its value and CBJ behaves again like BT, going to solution without any extra backtrack. Backjumping algorithms are usually not implemented in solvers. The improvement they provide becomes minor when they are combined with other types of improvement of BT, such as look ahead (see Section 7.2) and variable ordering heuristics (see Section 7.3).

Another type of look back technique is *nogood learning*. The idea is to store explanations of failure (*nogoods*) and then use them as new constraints to avoid exploring subtrees that subsume a nogood already met. This direction of research dates back to the '70s [Gaschnig, 1974; Dechter, 1986; Schiex and Verfaillie, 1993], where a nogood was essentially an instantiation that has led to a failure. Nogoods were more or less abandoned by the constraint reasoning community while the SAT community took the idea, improved it, and showed that nogoods can be useful to significantly decrease search effort (see Chapter 5 of this volume). Motivated by this success in SAT, the idea of nogood came back to the constraint reasoning community. Nogoods have been generalized to explanations of failure more expressive than simple assignments, and techniques from SAT have been used to process the nogoods [Katsirelos and Bacchus, 2003; Lecoutre et al., 2007]. Ohrimenko et al. [2009] have introduced *lazy clause generation*, a technique in which the Boolean variables of the clauses representing explanations of value removals are generated on the fly each time a domain is reduced by constraint propagation. This lazy generation prevents the solver from creating huge formulas with many variables that will never be used. For instance, consider the propagation of the constraint $x+y=z$, where $x,y \in \{0..5\}$ and $z \in \{0..20\}$. The domain of $z$ is reduced to $\{0..10\}$ by constraint propagation, and a clause $\neg[\![x \leq 5]\!] \vee \neg[\![y \leq 5]\!] \vee [\![z \leq 10]\!]$ is added to the SAT formula handling explanations. $[\![x \leq 5]\!]$, $[\![y \leq 5]\!]$, and $[\![z \leq 10]\!]$ are Boolean variables

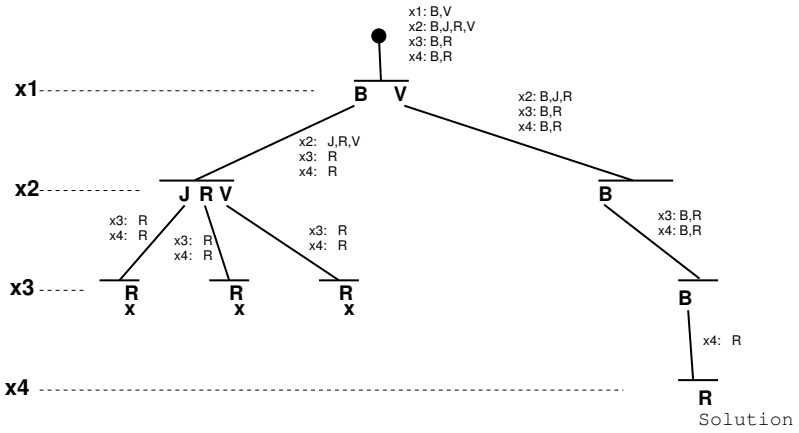


**Fig. 5** Search tree of forward-checking on the constraint network of Example 1. The new domains of non instantiated variables are given after each instantiation.

created on the fly the first time they appear in an explanation. Siala [2015] has used lazy clause generation in global constraints.

## *7.2 Look ahead*

Look ahead techniques apply constraint propagation at each node of the search tree explored by BT. The motivation is that it is better to detect an inconsistency once at the top of the search tree than an exponential number of times at the bottom of the tree. The levels of propagation that are used with BT are generally restricted to the removal of values in the domains. Higher levels of consistency are considered too expensive to be maintained at each node of the tree.

The first algorithm in this category is *forward-checking*, informally proposed by Golomb and Baumert [1965] and explicitly described and analyzed by Haralick and Elliott [1980] for binary constraints. Forward-checking is a search procedure in the same vein as BT. The difference with BT is that after every assignment of a variable $x_i$, forward-checking traverses the domain of all variables $x_k$ not yet instantiated and sharing a constraint $c$ with $x_i$, and removes values that violate constraint $c$. As soon as one of these variables $x_k$ has an empty domain, the current branch is abandoned and all domains are restored as before $x_i$'s assignment. With such an algorithm it is no longer necessary to check the compatibility of $x_i$'s values with already instantiated variables (see line 4 in Algorithm 1) because we have removed values of $x_i$ incompatible with already instantiated variables when $x_i$ was a future variable. Figure 5 displays the search tree traversed by forward-checking on the network of Example 1.

**Algorithme 3 :** Maintaining arc consistency (MAC)

```
function MAC (in N: network): Boolean
  begin
1   enforce AC on N
2   if a domain has been wiped out in N then return false
3   if all variables in N are ground then print D and return true
4   choose a variable x_i with |D(x_i)| > 1 and choose a value v_i in D(x_i)
5   if MAC(N|_{x_i=v_i}) then return true
6   return MAC(N|_{x_i≠v_i})
```

Deciding which level of propagation to apply at each node of the search tree has always been a concern in the CP community. More propagation means more work at each node but a smaller search tree to explore. Forward-checking has long been regarded as applying the right level of propagation. This can be explained by the fact that the problems on which we were testing the algorithms in the '80s/early '90s were often small and not very difficult. Since the mid-'90s, it is understood that the standard level of propagation to apply at each node is arc consistency. The idea of applying arc consistency after each variable assignment, though, dates back to Gaschnig [1974] and his algorithm CS-2, renamed DEEB in [Gaschnig, 1978], and later presented as *Really Full Look-ahead (RFL)* by Nadel [1988]. These algorithms perform what is called *d-way branching*. They behave like a BT procedure (see Algorithm 1) where each variable assignment is followed by enforcing arc consistency. Sabin and Freuder [1994] proposed *MAC (Maintaining Arc Consistency)*. MAC is an algorithm that not only enforces arc consistency after each assignment, but also after refuting the choice of a value which has led to a failure. In addition, MAC has the freedom to select another variable after the refutation step, even if the domain of the previously chosen variable has not yet been exhausted. This behavior is called 2-*way branching*. Most solvers are based on MAC.

MAC is presented in Algorithm 3. MAC first applies arc consistency on the constraint network (line 1). If none of the domains has been wiped out (line 2) and if we have not yet reached a solution (line 3), MAC selects a variable $x_i$ and a value $v_i$ for this variable, (line 4). The recursive call of line 5 enforces arc consistency on the network $N|_{x_i=v_i}$, where $x_i$ has been assigned value $v_i$, and continues the search on this subnetwork. If this branch leads to a failure (no solution in the subtree), the recursive call of line 6 propagates the refutation of $x_i = v_i$ by enforcing arc consistency on the network $N|_{x_i \neq v_i}$ and then continues the search in this new subtree. Figure 6 displays the search tree traversed by MAC on the network of Example 1. As we can see on the figure, MAC is backtrack-free. As soon as $x_1$ is assigned value $B$, a failure is detected when enforcing arc consistency. MAC branches on the refutation $x_1 \neq B$ and goes down directly to the solution in a single branch.

Lecoutre and Hemery [2007] have proposed AC3rm, an efficient algorithm for maintaining arc consistency during search. This algorithm stores supports like AC2001, uses multi-directionality of supports like AC7 [Bessiere et al., 1999], but

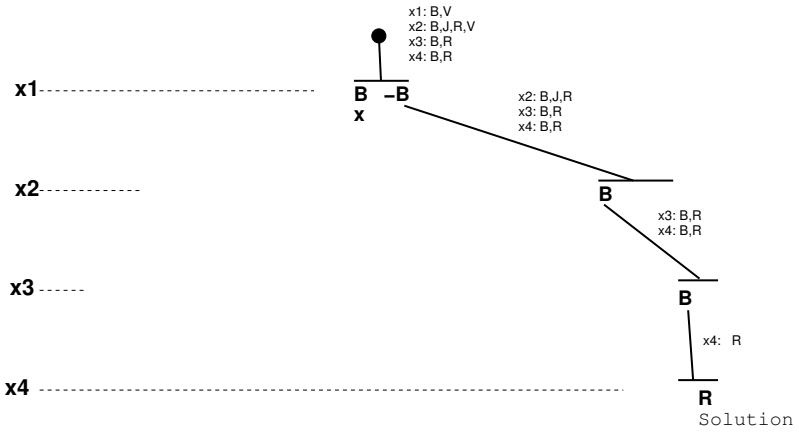


**Fig. 6** Search tree of MAC on the constraint network of Example 1. The new domains of non instantiated variables are given after each instantiation. "$-B$" denotes the removal of value $B$ from the domain.

does not restore supports when backtracking. It loses the optimality of AC2001, but it saves the cost of restoring supports.

## *7.3 Variable and value ordering heuristics*

Up to now, I have considered that the search procedure selects variables to instantiate and values to assign to them in lexicographic ordering. However, nothing prevents us from changing this arbitrary exploration ordering. In practice, it appears that the order in which we explore the search space can have a tremendous effect on the cpu time needed to find a solution.

### Variable ordering

Variable ordering heuristics can be classified as *static* or *dynamic*. Static variable orderings are computed before the search for solutions starts, and are kept unchanged during the whole search. They are generally based on criteria using the structure of the network. For instance, *maxdegree* selects variables in decreasing order of the number of constraints involving them. This promotes the detection of failures high in the search tree, which is less costly than detecting failures deep in the search tree.

Dynamic variable orderings take into account changes performed on the network at the current node, that is, in general, deletions of values from domains. Haralick and Elliott [1980] proposed the *mindom* heuristic, which selects as next variable the one that has the smallest number of values remaining in its domain. The motivation

is to minimize the branching degree in the tree. Bessiere and Régin [1996] proposed *dom/deg*, which combines information on the size of the domains with information on the structure of the network. *dom/deg* selects the variable with the smallest ratio size of the current domain over the number of constraints involving the variable.

Boussemart et al. [2004] proposed *dom/wdeg*, an *adaptive* variable ordering heuristic. In addition to the state of the network in the current node, *dom/wdeg* takes into account the *weighted degree* of variables. The weighted degree, inspired by SAT solvers, stores information about the failures that occurred in branches already explored by the search procedure. A counter $w_j$ stores the number of times a constraint $c_j$ has caused a failure, that is, the number of times the propagation of $c_j$ has caused an empty domain. The weighted degree of variable $x_i$ is the sum of the $w_j$'s for which $c_j$ involves $x_i$. At the beginning, $w_j = 1$ for all constraints. During search, each time $\texttt{Revise}(-, c_j)$ wipes out a domain, $w_j$ is incremented by 1. The heuristic *dom/wdeg* chooses the variable $x_i$ which minimizes the ratio size of the current domain over weighted degree. This heuristic starts like *dom/deg*, but during search, it focuses more and more on the variables involved in constraints that are difficult to satisfy. *dom/wdeg* is a very good variable ordering heuristic on many kinds of problems.

Lecoutre et al. [2006] have also proposed the *last conflict reasoning* technique. This is not exactly a variable ordering heuristic. This is a technique that is combined with a variable ordering heuristic to mimic backjumping by simply forcing the choice of the next variable to instantiate after a dead-end.

### Value ordering

Value ordering heuristics have led to many less contributions than variable ordering heuristics because the impact on the performance of the solver is much lower. One reason is that as soon as a problem is difficult to solve, it is very likely that the solver will spend the greater part of the time exploring large inconsistent subproblems. On inconsistent subproblems, the order in which we select values in a domain has little effect because we must try them all. Recently, Mehta et al. [2011] have shown that value ordering could have an effect when using an adaptive variable ordering heuristic. As adaptive variable ordering heuristics are affected by past operations of the search procedure, selecting a value instead of another can have an effect on the next choice of variable.

### Search heuristics

There also exist heuristics that guide the exploration of the search tree by selecting at the same time the variable and the value to assign. Refalo [2004] proposed *impact-based* heuristic, a heuristic that selects the variable and value with the highest impact, where impact measures the importance of an assignment in reducing the search space. Michel and Van Hentenryck [2012] have proposed *activity-based*

*search*, a heuristic that selects the variable and value with the highest activity, where activity measures by the number of times the domains of variables are reduced by constraint propagation.

# 8 Other Techniques to Improve Search

## *8.1 Non-standard backtracking search*

On some kinds of CSPs it may happen that backtracking search takes enormous amount of time to find a solution despite the problem has many solutions and most other orders for selecting values for variables would have led to a solution very quickly. The reason for this unexpected behavior is that the value ordering heuristic has told the search procedure to perform a variable assignment that has led the search into a huge inconsistent subtree. This phenomenon is called *heavy-tailed* behavior [Gomes et al., 1997, 2005].

Harvey and Ginsberg [1995] have made the twofold observation that heuristics have more chances to take wrong decisions (i.e., variable assignments leading to an inconsistent subtree) early in the search when little is known about the problem, and that wrong decisions high in the tree are critical because they can lead to huge inconsistent subtrees. To address these two issues, Harvey and Ginsberg have proposed *limited discrepancy search (LDS)*. A *discrepancy* is a node at which the search procedure does not select the value proposed by the heuristic. When reaching a non-solution leaf (that is, a failure or inconsistent node), instead of just undoing the last assignment, LDS tries to make a discrepancy as high as possible in the search tree. LDS iteratively explores the (unique) branch with zero discrepancies, then all branches with at most one discrepancy –starting from discrepancies higher in the tree–, then all branches with at most two discrepancies, and so on until a solution is found or the whole tree has been explored. Figure 7 displays the shape of a search tree explored by a backtracking search procedure (it could be BT, forward-checking, etc.). We take the usual convention that backtracking search visits the leaves from left to right. The order in which leaves are visited by LDS on the same search tree is given for each leaf visited until finding a solution. We observe that the solution found by LDS (at its 8th leaf visited) is not the same as the solution found by a standard backtracking search. Backtracking search would find first the leftmost solution of the tree (at the 12th leaf visited).

The main weakness of LDS is that it can visit several times the same leaf, leading to high redundancy in search. See for instance the first leaf on the left of the search tree in Figure 7. It is visited twice before finding a solution: once during the pass with zero discrepancies and once during the pass with at most one discrepancy. Korf [1996] has proposed *improved limited discrepancy search (ILDS)* to fix this drawback. Unfortunately, this improvement is done at the price of performing discrepancies low in the tree before discrepancies high in the tree. Now, value or-

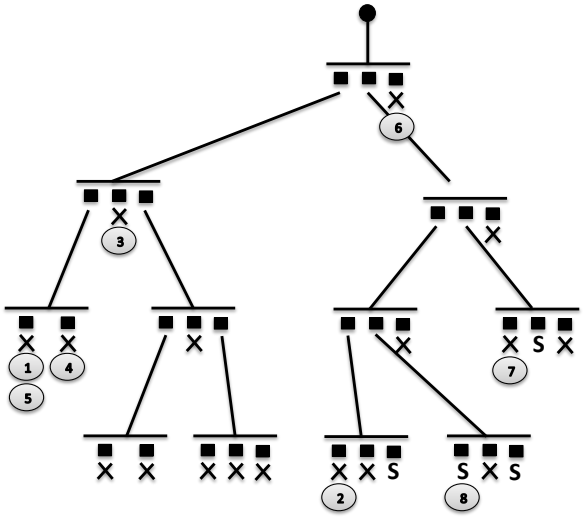


**Fig. 7** Order of visit of the leaves of a search tree explored by LDS. Squares denote the nodes. "x" denotes failure nodes and "S" denotes solution nodes.

dering heuristics tend to be less informed and make more mistakes early in the tree than later, when they are more informed. Walsh [1997] has proposed *depth-bounded discrepancy search (DDS)*. Like ILDS, DDS explores the branch with zero discrepancies, then the branches with exactly one discrepancy, and so on, and thus DDS visits at most once each leaf. However, like LDS and ILDS, DDS can expand many times non-leaf nodes. A non-leaf node is expanded by DDS as often as there are leaves under it that DDS needs to reach with different numbers of discrepancies. For a CSP solving procedure, expanding several times a non-leaf node means assigning several times the same variables with the same values and redoing several times the same constraint propagation. In parallel to DDS, Meseguer [1997] has proposed *interleaved depth-first search (IDFS)*. IDFS does not have the drawback of redundancy of expanding non-leaf nodes because each time it expands a node, it stores the current state of the search for later visits. As this causes exponential space, Meseguer proposed a limited version that stores the current state only for a limited number of levels in the tree, making sequential search in the lower levels.

*Restart* strategy is another technique that allows backtracking search to escape from early wrong decisions and it is much simpler to implement. Adding the restart strategy to a backtracking search procedure is done by counting the number of nodes visited or the number of failures reached, and when this number is equal to a given threshold $\theta$, we restart the search, that is, the backtracking search is reset. Of course, in order not to re-explore exactly the same (inconsistent) subtree, we need to put some randomness in the value ordering heuristic. If the threshold $\theta$ at which we reset is constant, the risk is that it could be too small compared to the inherent

difficulty of the problem, and none of the runs has chances to find a solution before $\theta$ nodes/failures. This issue is easily fixed by initializing $\theta$ to a very small value, and at each restart, we change its value. The most common strategy for updating $\theta$ is to follow a geometric progression, that is, multiplying $\theta$ by a given constant – usually 1.5– each time we restart search. Another updating strategy, more popular in SAT than in CSPs, is to make $\theta$ follow a *Luby* sequence [Luby et al., 1993]. Luby's sequences grow very slowly. They have shown very good performance.

## *8.2 Large neighborhood search*

Some CSPs, whereas not being inherently difficult to solve, are so large in size that they cannot be solved in reasonable time by backtracking search, even when combined with improvements that we have seen in Sections 7 or 8.1. An alternative that is often used (especially when the goal is to find a *good* solution, i.e., an assignment *maximizing* an objective criterion) is to relax the completeness of backtracking search and to go for a more local-search style algorithm. However, pure local search is known to have poor performance on CSPs encoding real problems. Shaw [1998] has proposed *large neighborhood search (LNS)* to combine the best of local search and of backtracking search with constraint propagation. Like standard local search, LNS starts with a complete assignment of the variables. If this assignment is not solution (or not a good enough assignment), LNS selects a set $S$ of variables to "relax", that is, to be reset to their initial domain. The variables not in $S$ remain assigned to their value in the current assignment. The set $S$ defines the neighborhood to be explored. Then, LNS uses backtracking search and constraint propagation to "repair" the partial assignment, that is, to assign a new value to the variables in $S$. The loop relax/repair is repeated until a solution is found or a time limit has been reached. An essential component of the success of LNS is how to select the set $S$ of variables to relax. If $S$ is too large, finding a new assignment depends more on backtracking search and constraint propagation than on the neighborhood and we fall in the same drawback as backtracking search. If $S$ is too small, backtracking search may not have enough space to explore and may not be able to find good assignments. Shaw proposes to increase the size of $S$ when several executions of the loop relax/repair do not improve the assignment. If the size of $S$ grows until the number of variables in the CSP, completeness of the search process is guaranteed. Concerning the choice of which variables to put in $S$, good strategies are usually specific to the type of problem been solved.

## *8.3 Symmetries*

When modeling a problem as a CSP, it is often the case that several variables are defined to represent several occurrences of the same object. For instance, two vari-

ables $x_i$ and $x_j$ can represent the two history courses that a class must have in the week. If variable $x_i$ takes value Monday-8am and $x_j$ value Thursday-9am, or the reverse, it will not change the fact that this instantiation is solution or not. We then say that $x_i$ and $x_j$ are symmetrical. The same can happen for values. The purpose of symmetry detection techniques is to prevent the search procedure from exploring two subtrees that are symmetrical. These techniques may cut branches of the search tree that contain solutions but only if these solutions have a symmetrical solution in another part of the search tree. Freuder [1991] raised the problem of symmetry of values by defining *interchangeability* of values. Cooper [1997] extended this notion to *substitutability* of values. Since then, many works have studied many sorts of symmetries [Benhamou, 1994; Gent et al., 2006].

Another approach to reduce the symmetries generated when modeling a problem is to use *set variables* instead of integer variables. A set variable is a variable that will be assigned a *set* of values instead of one single value. In our example above, $x_i$ and $x_j$ could be replaced by a single set variable $s$ representing the history courses. A constraint $|s| = 2$ should be added to ensure that $s$ will be assigned two values —the two history courses in the week. Letting variables taking sets of values instead of single values gives rise to a number of possible assignments exponential in the number of possible values. Thus, domains of set variables are not represented in extension. They are represented by two bounds. Gervet [1994] represents the domain of a set variable $s$ by a lower bound $lb(s)$ (the mandatory values) and an upper bound $ub(s)$ (the possible values). If $lb(s) = \{1,3\}$ and $ub(s) = \{1,2,3,4\}$, $s$ has a domain of possible assignments equal to $\{(1,3),(1,2,3),(1,3,4),(1,2,3,4)\}$. Other representations of the domains of set variables have been proposed, such as *length-lex*, which stores as bounds the smallest possible set and the greatest possible set of the set variable, where sets are ordered by increasing cardinality, ties being broken lexicographically [Gervet and Van Hentenryck, 2006].

## 9 Global Constraints

When modeling real problems into constraint networks, we observe that some types of requirements (aka *patterns*) on the combinations of values that sets of variables can take occur in various problems. For instance, it is often necessary to express the fact that a set of variables must all take different values. The size of the set of variables is not the same in all problems. The constraint `alldifferent`, which expresses this pattern, can involve any number of variables. This kind of constraints, defined by a Boolean function that can take any number of variables as parameter, are called *global constraints*. Beldiceanu et al. [2005b] have built a catalog of more than 400 global constraints.

**Example 8** The global constraint $\texttt{atleast}_{p,v}(x_1,\ldots,x_n)$ is defined on any sequence of $n$ variables, $n \geq 1$, such that at least $p$ variables in $x_1,\ldots,x_n$ are equal to $v$. ♢

**Example 9** The global constraint $\texttt{alldifferent}(x_1,\ldots,x_n)$ is defined on any sequence of $n$ variables, $n \geq 2$, such that $x_i \neq x_j$ for all $i,j,1 \leq i,j \leq n, i \neq j$. $\diamondsuit$

**Example 10** The global constraint $\texttt{NValue}(y,x_1,\ldots,x_n)$ is defined on any sequence of $n+1$ variables, $n \geq 1$, such that $|\{x_i \mid 1 \leq i \leq n\}| = y$. $\diamondsuit$

If a global constraint is available in a solver, the user can easily express the associated pattern, which could otherwise be complex to express. These constraints may involve a large number of variables. It is thus important to have a way to propagate them other than a generic arc consistency algorithm such as the function `Revise` of AC3 or AC2001. Bear in mind that the optimal generic algorithms for enforcing arc consistency are in $O(erd^r)$ for constraint networks with constraints involving $r$ variables (see Section 4.1).

It is not always possible to design an efficient arc consistency *propagator* (i.e., algorithm enforcing arc consistency) for a global constraint. Bessiere et al. [2007] have shown that if the problem of the existence of a valid tuple satisfying the global constraint is NP-complete, then there does not exist any polynomial time arc consistency propagator for this global constraint, unless P = NP. This is the case of the global constraint `NValue`, for which it is thus useless to look for an arc consistency propagator. On such global constraints, we usually enforce bounds consistency.

Even if we do not consider the constraints for which arc consistency is NP-hard, designing arc consistency propagators for all the remaining constraints in the catalog is by far too cumbersome.

A solution to propagate a global constraint without designing a propagator is to decompose it into simpler constraints. Decomposing a global constraint means replacing each of its instances by a subnetwork of bounded-arity constraints (and new variables if necessary). The subnetwork has to be polynomial in the size of the original variables and domains and it has to preserve the set of allowed tuples on the variables of the original constraint. More formally, a decomposition of a global constraint $G$ is a *polynomial* transformation $\delta_k$ ($k$ being an integer), which for any network $N = (X(c), D, \{c\})$ where $c$ is an instance of $G$ of arity $|X(c)|$, returns a network $\delta_k(N) = (X_{\delta_k(N)}, D_{\delta_k(N)}, C_{\delta_k(N)})$ such that $X(c) \subseteq X_{\delta_k(N)}$, $D(x_i) = D_{\delta_k(N)}(x_i)$ for all $x_i \in X(c)$, $|X(c_j)| \leq k$ for all $c_j \in C_{\delta_k(N)}$, and $sol(N)$ is equal to the projection of $sol(\delta_k(N))$ on $X(c)$.

**Example 11** $\texttt{atleast}_{p,v}(x_1,\ldots,x_n)$ can be decomposed with $n+1$ additional variables $y_0,\ldots,y_n$. The transformation contains the ternary 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 domains $D(y_0) = \{0\}, D(y_n) = \{p,\ldots,n\}$ and $D(y_i) = \{0,\ldots,n\}$ for all $i, 1 \leq i < n$. $\diamondsuit$

The central issue is of course to find decompositions that *preserve* arc consistency on the original constraint. That is, for any instance $c$ of the global constraint $G$ on any initial domain $D$ on $X(c)$, we want that for every domain $D'$ included in $D$, enforcing arc consistency on the decomposition removes from $D'$ the same values as arc consistency on $c$. The decomposition of $\texttt{atleast}_{p,v}$ given in Example 11 preserves arc consistency because it has a Berge-acyclic structure. Many constraints among the 400 of the catalog allow a decomposition that preserves arc consistency. Global

constraints as important as `Regular` [Pesant, 2004] allow such a decomposition [Beldiceanu et al., 2005a; Bessiere et al., 2008]. However, Bessiere et al. [2009] have shown that if a constraint subsumes a Boolean function non representable by a monotone Boolean circuit of polynomial size then it does not admit any decomposition preserving arc consistency, even if computing arc consistency on this constraint is polynomial time. For such constraints, the only option is to implement a specific propagator. A famous example is the constraint `alldifferent`. Régin [1994] used the problem of maximum matching in a bipartite graph to produce an arc consistency propagator for the constraint `alldifferent`. Another popular constraint that cannot be decomposed and thus requires a specific propagator is the `global-cardinality` constraint [Régin, 1996]. The non-decomposability result of [Bessiere et al., 2009] has a corollary that goes beyond the CP world. It implies that there does not exist any SAT model of polynomial size that mimics arc consistency for non decomposable global constraints. An illustration is the *pigeon-hole* problem,[4] solved in milliseconds by a CP solver with an `alldifferent` constraint on the pigeons, and not solved in hours by the best SAT solvers as soon as the number of pigeons is above 20.

When modeling a problem, finding the right global constraint among more than 400 in the catalog can be difficult. Beldiceanu and Simonis [2011] have proposed *ConstraintSeeker*, a system that takes as input a few tuples satisfying (or violating) the pattern we want to express and returns an ordered set of candidate constraints.

## 10 Conclusion and New Trends

In this chapter, I have presented various techniques for solving the CSP. When solving real problems with constraint reasoning, we quickly realize that answering yes or no, or just giving a solution to the specified network is often not sufficient to satisfy the user. The user may have forgotten to express some constraints that will come to her mind when she will see a solution that is not satisfactory (e.g., a schedule with four hours of math course in a row). On the contrary, the user may have put too many constraints to express her preferences and she will have to relax some to make the network satisfiable (e.g., relax the constraint on the end of the courses at 4pm). A first approach to deal with this issue was the dynamic CSP [Dechter and Dechter, 1988], where the user interacts with the solver by adding or removing constraints according to the solution provided (or not) by the solver. The maxCSP [Freuder and Wallace, 1992] and the valued CSP [Schiex et al., 1995] can also handle this issue because they look for solutions that are optimal according to a given criterion (e.g., maximum number of satisfied constraints for maxCSP). They will be presented in Chapter 7 of this volume. The problem to be solved may also contain uncertainties about the values of some of the variables that the user does not decide. (For instance, variables related to the weather.) In such a context, a solu-

[4] The pigeon-hole problem is to assign $n$ pigeons to $n-1$ holes in such a way that no hole contains more than one pigeon.

tion must satisfy the constraints, whatever the values taken by these variables, or it should maximize a probability of satisfaction. The mixed [Fargier et al., 1996] or stochastic CSPs [Walsh, 2002] deal with this case. The problem can be distributed on several remote places, communicating by messages and requiring that some data are kept private. In such cases, we use distributed CSPs [Yokoo et al., 1998]. In some problems, we may need to represent information that is not easy to discretize. In this case, we use numerical CSPs, in which variables take their values in domains composed of intervals on the reals [Benhamou and Granvilliers, 2006]. Finally, on some problems, it can happen that we do not have all the values of the domains at the beginning of the solving process, or the scope of a constraint will depend on the value of other variables. Open CSPs allow such missing information [Faltings and Macho-Gonzalez, 2005]. There are many other extensions of the CSP framework that are not yet central but could become important if their interest is confirmed.

CP is now a mature field. It is thus simultaneously increasingly difficult to make fundamental contributions at the core of CP, and increasingly easy to show the practical usefulness of CP in other areas of science and industry. Data science is an example of the use of CP in other areas. The international conference on constraint programming organizes a special track on CP and data science, and the *Artificial Intelligence* journal has recently published a special issue on "Combining Constraint Solving with Mining and Learning". In these two venues, we find papers using CP for data analysis. For instance, CP can be used for solving pattern mining, sequence mining, clustering, and other data mining problems [Ugarte et al., 2017; Dao et al., 2017; Schaus et al., 2017]. (The first CP model for pattern mining had been proposed by Guns et al. [2011].) If CP can be used for data analysis, data analysis can itself be used to help building CP models. When building a CP model for a combinatorial problem, it can happen that the domain expert is not able to accurately describe her problem. When data from the problem are available, Lombardi et al. [2017] have proposed to extract missing or incomplete parts of the CP model by learning decision trees or artificial neural networks from these data. They embed these learned decision trees or artificial neural networks into the CP model either as constraints or as objective functions. This technique was applied to thermal-aware workload dispatching. In the same vein, Bessiere et al. [2017] have proposed the inductive constraint programming loop. Solutions of a combinatorial problem (e.g., scheduling external visits in an hospital, energy-aware data center) are executed, and their behavior is observed. Depending on the efficiency of this execution (e.g., waiting time for patients, actual price paid for electricity), a machine learning component learns new constraints to revise the model. New solutions are computed, executed, observed, and so on.

## Acknowledgments

I am very grateful to Anastasia Paparrizou and Thomas Schiex for their thorough reading of this chapter and for their suggestions. I also thank Clément Carbonnel,

Michele Lombardi, and Michela Milano for their help in describing some of the contributions presented in this chapter.

## References

Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency restoration and explanations in dynamic csps -application to configuration. *Artificial Intelligence*, 135:199–234.

Andersen, H., Hadzic, T., Hooker, J. N., and P.Tiedemann (2007). A constraint store based on multivalued decision diagrams. In *Proceedings CP'07*, pages 118–132, Providence RI.

Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Arnborg, S. (1985). Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23.

Beldiceanu, N., Carlsson, M., Debruyne, R., and Petit, T. (2005a). Reformulation of global constraints based on constraints checkers. *Constraints*, 10(4):339–362.

Beldiceanu, N., Carlsson, M., and Rampon, J. (2005b). Global constraint catalog. Technical Report T2005:08, Swedish Institute of Computer Science, Kista, Sweden.

Beldiceanu, N. and Simonis, H. (2011). A constraint seeker: Finding and ranking global constraints from examples. In *Proceedings CP'11*, pages 12–26, Perugia, Italy.

Benhamou, B. (1994). Study of symmetry in constraint satisfaction problems. In *Proceedings PPCP'94*, pages 249–257, Seattle WA.

Benhamou, F. and Granvilliers, L. (2006). Continuous and interval constraints. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 16. Elsevier.

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

Bergman, D., Ciré, A. A., and van Hoeve, W. J. (2014). MDD propagation for sequence constraints. *Journal of Artificial Intelligence Research*, 50:697–722.

Berlandier, P. (1995). Improving domain filtering using restricted path consistency. In *Proceedings IEEE-CAIA'95*, Los Angeles CA.

Bertelé, U. and Brioshi, F. (1972). *Nonserial Dynamic Programming*. Academic Press.

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

Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., and Walsh, T. (2008). Slide: A useful special case of the cardpath constraint. In *Proceedings ECAI'08*, pages 475–479, Patras, Greece.

Bessiere, C., Hebrard, E., Hnich, B., and Walsh, T. (2007). The complexity of reasoning with global constraints. *Constraints*, 12(2):239–259.

Bessiere, C., Katsirelos, G., Narodytska, N., and Walsh, T. (2009). Circuit complexity and decompositions of global constraints. In *Proceedings IJCAI'09*, pages 412–418, Pasadena CA.

Bessiere, C., Raedt, L. D., Guns, T., Kotthoff, L., Nanni, M., Nijssen, S., O'Sullivan, B., Paparrizou, A., Pedreschi, D., and Simonis, H. (2017). The inductive constraint programming loop. *IEEE Intelligent Systems*, 32(5):44–52.

Bessiere, C. and Régin, J. (1996). MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings CP'96*, pages 61–75, Cambridge MA.

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

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

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings ECAI'04*, pages 146–150, Valencia, Spain.

Bulatov, A. A. (2017). A dichotomy theorem for nonuniform CSPs. In *Proceedings IEEE-FOCS'17*, pages 319–330, Berkeley, CA.

Bulatov, A. A. and Dalmau, V. (2006). A simple algorithm for mal'tsev constraints. *SIAM J. Comput.*, 36(1):16–27.

Cabon, C., de Givry, S., Lobjois, L., Schiex, T., and Warners, J. (1999). Radio link frequency assignment. *Constraints*, 4:79–89.

Carbonnel, C. (2016). The dichotomy for conservative constraint satisfaction is polynomially decidable. In *Proceedings CP'16*, pages 130–146, Toulouse, France.

Chen, H., Dalmau, V., and GruSien, B. (2013). Arc consistency and friends. *J. Log. Comput.*, 23(1):87–108.

Cheng, B. M. W., Lee, J. H., and Wu, J. C. K. (1997). A nurse rostering system using constraint programming and redundant modeling. *IEEE Trans. Information Technology in Biomedicine*, 1(1):44–54.

Cohen, D. A. and Jeavons, P. G. (2017). The power of propagation: when GAC is enough. *Constraints*, 22(1):3–23.

Cooper, M. (1997). Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90:1–24.

Cooper, M., Cohen, D., and Jeavons, P. (1994). Characterising tractable constraints. *Artificial Intelligence*, 65:347–361.

Dao, T., Duong, K., and Vrain, C. (2017). Constrained clustering by constraint programming. *Artificial Intelligence*, 244:70–94.

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

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

Dechter, R. (1986). Learning while searching in constraint satisfaction problems. In *Proceedings AAAI'86*, pages 178–183, Philadelphia PA.

Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312.

Dechter, R. and Dechter, A. (1988). Belief maintenance in dynamic constraint networks. In *Proceedings AAAI'88*, pages 37–42, St Paul MN.

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

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

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

Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.-C., and Schaus, P. (2016). Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *Proceedings CP'16*, pages 207–223, Toulouse, France.

Deville, Y., Barette, O., and Van Hentenryck, P. (1999). Constraint satisfaction over connected row convex constraints. *Artificial Intelligence*, 109(1-2):243–271.

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

Faltings, B. and Macho-Gonzalez, S. (2005). Open constraint programming. *Artificial Intelligence*, 161(1-2):181–208.

Fargier, H., Lang, J., and Schiex, T. (1996). Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge. In *Proceedings AAAI'96*, pages 175–180, Portland OR.

Feder, T. and Vardi, M. (1998). The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM J. Comput.*, 28(1):57–104.

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

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

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

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

Freuder, E. (1990). Complexity of k-tree structured constraint satisfaction problems. In *Proceedings AAAI'90*, pages 4–9, Boston MA.

Freuder, E. (1991). Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings AAAI'91*, pages 227–233, Anaheim CA.

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

Freuder, E. and Wallace, R. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70.

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

Gaschnig, J. (1978). Experimental case studies of backtrack vs waltz-type vs new algorithms for satisficing assignment problems. In *Proceedings Second Canadian Conference on Artificial Intelligence*, pages 268–277, Toronto, Canada.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical Report Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh PA.

Gaspin, C. and Westhof, E. (1994). The determination of the secondary structures of ribonucleic acids as a constraint satisfaction problem. In *IEEE Colloquium on Molecular BioInformatics*, London, UK.

Gent, I., Petrie, K., and Puget, J. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 10. Elsevier.

Gervet, C. (1994). Conjunto: Constraint logic programming with finite set domains. In *Proceedings ILPS'94*, pages 339–358, Ithaca NY.

Gervet, C. and Van Hentenryck, P. (2006). Length-lex ordering for set csps. In *Proceedings AAAI'06*, pages 48–53, Boston MA.

Golomb, S. and Baumert, L. (1965). Backtrack programming. *Journal of the ACM*, 12(4):516–524.

Gomes, C., Fernandez, C., Selman, B., and Bessiere, C. (2005). Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337.

Gomes, C., Selman, B., and Crato, N. (1997). Heavy-tailed distributions in combinatorial search. In *Proceedings CP'97*, pages 121–135, Linz, Austria.

Gottlob, G., Leone, N., and Scarcello, F. (2000). A comparison of structural csp decomposition methods. *Artificial Intelligence*, 124(2):243–282.

Guns, T., Nijssen, S., and Raedt, L. D. (2011). Itemset mining: A constraint programming perspective. *Artificial Intelligence*, 175(12-13):1951–1983.

Gyssens, M., Jeavons, P., and Cohen, D. (1994). Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89.

Haralick, R. and Elliott, G. (1980). Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313.

Harvey, W. and Ginsberg, M. (1995). Limited discrepancy search. In *Proceedings IJCAI'95*, pages 607–613, Montréal, Canada.

Hoda, S., van Hoeve, W. J., and Hooker, J. N. (2010). A systematic approach to mdd-based constraint programming. In *Proceedings CP'10*, pages 266–280, St Andrews, Scotland.

ILOG (1997). *User's manual*. ILOG Solver, 4.0 edition.

Janssen, P., Jégou, P., Nouguier, B., and Vilarem, M. C. (1989). A filtering process for general constraint-satisfaction problems: Achieving pairewise-consistency using an associated binary representation. In *Proceedings IEEE-ICTAI'89*, pages 420–427, Fairfax VA.

Janssen, P. and Vilarem, M. (1988). Problèmes de satisfaction de contraintes : techniques de résolution et application à la synthèse de peptides. Technical Report 54, CRIM, University of Montpellier, France.

Jeavons, P., Cohen, D., and Cooper, M. (1998). Constraints, consistency and closure. *Artificial Intelligence*, 101:251–265.

Jeavons, P. and Cooper, M. (1995). Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2):327–339.

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

Jégou, P. and Terrioux, C. (2003). Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1):43–75.

Katsirelos, G. and Bacchus, F. (2003). Unrestricted nogood recording in csp search. In *Proceedings CP'03*, pages 873–877, Kinsale, Ireland.

Korf, R. (1996). Improved limited discrepancy search. In *Proceedings AAAI'96*, pages 286–291, Portland OR.

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

Lecoutre, C. (2011). STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371.

Lecoutre, C. and Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings IJCAI'07*, pages 125–130, Hyderabad, India.

Lecoutre, C., Likitvivatanavong, C., and Yap, R. H. C. (2015). STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2006). Last conflict based reasoning. In *Proceedings ECAI'06*, pages 133–137, Riva del Garda, Italy.

Lecoutre, C., Sais, L., Tabary, S., and Vidal, V. (2007). Nogood recording from restarts. In *Proceedings IJCAI'07*, pages 131–136, Hyderabad, India.

Lhomme, O. (1993). Consistency techniques for numeric CSPs. In *Proceedings IJCAI'93*, pages 232–238, Chambéry, France.

Lombardi, M., Milano, M., and Bartolini, A. (2017). Empirical decision model learning. *Artificial Intelligence*, 244:343–367.

Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180.

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

Mackworth, A. (1977b). On reading sketch maps. In *Proceedings IJCAI'77*, pages 598–606, Cambridge MA.

Marquis, P., Papini, O., and Prade, H., editors (2014). *Panorama de l'intelligence artificielle : Ses bases methodologiques ses developpements*. Cépaduès.

Marquis, P., Papini, O., and Prade, H., editors (2020). *A Guided Tour of Artificial Intelligence Research, Volume II: AI Algorithms*. Springer.

Martin, P. and Shmoys, D. (1996). A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings IPCO'96*, pages 389–403, Vancouver, BC.

Mehta, D., O'Sullivan, B., and Quesada, L. (2011). Value ordering for finding all solutions: Interactions with adaptive variable ordering. In *Proceedings CP'11*, pages 606–620, Perugia, Italy.

Meseguer, P. (1997). Interleaved depth-first search. In *Proceedings IJCAI'97*, pages 1382–1387, Nagoya, Japan.

Michel, L. and Van Hentenryck, P. (2012). Activity-based search for black-box constraint programming solvers. In *Proceedings CPAIOR'12*, pages 228–243, Nantes, France.

Mohr, R. and Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233.

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

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

Nadel, B. (1988). 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.

Ohrimenko, O., Stuckey, P., and Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, 14(3):357–391.

Pesant, G. (2004). A regular language membership constraint for finite sequences of variables. In *Proceedings CP'04*, pages 482–495, Toronto, Canada.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299.

Refalo, P. (2004). Impact-based search strategies for constraint programming. In *Proceedings CP'04*, pages 557–571, Toronto, Canada.

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

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

Sabin, D. and Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP'94*, Seattle WA.

Schaus, P., Aoga, J. O. R., and Guns, T. (2017). Coversize: A global constraint for frequency-based itemset mining. In *Proceedings CP'17*, pages 529–546, Melbourne, Australia.

Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems: hard and easy problems. In *Proceedings IJCAI'95*, pages 631–637, Montréal, Canada.

Schiex, T. and Verfaillie, G. (1993). Nogood recording for static and dynamic CSPs. In *Proceedings IEEE-ICTAI'93*, pages 48–55, Boston MA.

Shaw, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings CP'98*, pages 417–431, Pisa, Italy.

Siala, M. (2015). Search, propagation, and learning in sequencing and scheduling problems. *Constraints*, 20(4):479–480.

Ugarte, W., Boizumault, P., Crémilleux, B., Lepailleur, A., Loudni, S., Plantevit, M., Raïssi, C., and Soulet, A. (2017). Skypattern mining: From pattern con-

densed representations to dynamic constraint satisfaction problems. *Artificial Intelligence*, 244:48–69.

Ullmann, J. (2007). Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678.

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

Walsh, T. (1997). Depth-bounded discrepancy search. In *Proceedings IJCAI'97*, pages 1388–1393, Nagoya, Japan.

Walsh, T. (2002). Stochastic constraint programming. In *Proceedings ECAI'02*, pages 111–115, Lyons, France.

Waltz, D. (1972). Generating semantic descriptions from drawings of scenes with shadows. Tech.Rep. MAC AI-271, MIT.

Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1998). The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685.

Zhuk, D. (2017). A proof of CSP dichotomy conjecture. In *Proceedings IEEE-FOCS'17*, pages 331–342, Berkeley, CA.