Towards 40 Years of Constraint Reasoning [*]

Pedro Meseguer

IIIA - CSIC, Campus Universitat Autònoma de Barcelona
08193 Bellaterra, Spain.
pedro@iiia.csic.es

Abstract. Research on constraints started in early 70's. We are approaching 40 years since the beginning of this successful field, and it is an opportunity to revise what has been reached. This paper is a personal view of the accomplishments in this field. We summarize the main achievements along three dimensions: constraint solving, modelling and programming. We devote special attention to constraint solving, covering popular topics such as search, inference (especially arc consistency), combination of search and inference, symmetry exploitation, global constraints and extensions to the classical model. For space reasons, several topics have been deliberately omitted.

1 Introduction

In 1970, Richard Fikes published a paper –partially based on his recently presented PhD thesis– on how to solve some problems using constraint satisfaction [35]. A bit later, David Waltz, a MIT PhD student working in vision introduced some processing based on constraints to understand the sense of edges in 3D images [95]. In 1974 Ugo Montanari published a paper [71], setting the basic concepts of constraint processing. In 1977, Alan Mackworth presented several algorithms to achieve basic forms of local consistency [62], some of which are still in use. These publications can be seen as the seminal works setting the basis of Constraint Reasoning, a new field at that time which has been very flourishing and successful until today. After the initial years, where these topics where mainly in academic workshops, this new field matured producing many publications in major AI conferences, having its own conferences and setting its own journals. In parallel, constraints were introduced in the field of logic programming, producing a new approach named constraint logic programming. Some companies started selling software based on constraints. The term *constraint programming* was coined, independently of the underlying programming style. Continuous improvements in constraints solving algorithms have caused substantial improvements in performance, while different extensions to the basic constraint model have significatively enlarged its expressivity. Now, we are closer to 40 years of this starting point, and it is meaningful to revise what has been done in this field. This is the purpose of this paper, which is necessarily short and leaves many topics untouched. The interested reader may consult some of the books existing on this topic [50, 89, 63, 28, 1, 78] for a more detailed historical perspective, as well as for a much broader and deeper description of its different aspects.

[*] Supported by the Spanish project TIN2009-13591-C02-02.

This new field took constraints as its central element; a *constraint* on a set of variables permits some value combinations while forbids others (assuming a finite number of possible assignments for variables). The task of interest was (and remains) finding an assignment of values to variables satisfying all constraints. Many problems of combinatorial nature can be formulated in terms of a finite number of variables and constraints, abstracting details that are application-dependent. This caused a lot of interest, because many real-world problems could benefit from this new technology (for instance, planning and scheduling, time-tabling, vehicle routing, configuration, and different applications in networking and bioinformatics, to name a few).

In this paper, we report the main accomplishments in Constraint Reasoning along three dimensions: constraint solving, constraint modelling and constraint programming. We devote substantially more space to constraint solving because it is the topic on which more work has been done by the research community working on constraints. We describe popular topics such as search (and in particular the well-known backtracking algorithm), inference (complete and incomplete, in this last class falls the arc-consistency idea), combinations of search and incomplete inference (forward checking and MAC algorithms), symmetry exploitation, global constraints and extensions to the classical constraint network model.

A personal view in these almost 40 years of history considers three consecutive periods, which do not have sharp boundaries but they are recognizable considering the dominant research topics in the community. They are:

- Initially, researchers were mainly interested in search procedures (including heuristics to improve search performance) in the context of binary constraints. Another main topic of interest was local consistency, mainly considered as preprocessing before search.
- Interests evolved towards more sophisticated search procedures, many of them maintaining some form of local consistency during search. Constraints were no longer assumed to be binary, and global constraints appeared. The role of inference methods was properly understood. Constraint modelling started to attract attention. Initial constraint programming environments were available.
- In the last decade, we have seen considerable performance improvements on symmetry exploitation, new global constraints and decompositions. Some extensions to the classical model have been proposed: continuous domains, soft constraints, distributed environments, etc. Connections with related fields, such as Operations Research or SAT, have been clarified. Constraint modelling and programming have reached some maturity.

The rest of this paper is organized as follows. First, we define the problem in Section 2, providing some classical examples. Second, we summarize the main solving approaches, grouped in the three main families of search, inference and hybrids (combining search and inference) in Section 3. Algorithmic improvements and extensions to the classical model are also described in Section 3. Third, we consider the issue of constraint modelling, that is, given a real-world problem producing a constraint model for it in Section 4. Fourth, we describe different approaches to implement the constraint model into a constraint programming environment in Section 5. Finally, we conclude the paper in Section 6, mentioning those topics which have been deliberately omitted. Because space limitations, descriptions are necessarily short.

2 The Problem

A *constraint network* is defined by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{x_1, x_2, ..., x_n\}$ is a set of n variables taking values in a collection of finite and discrete domains $\mathcal{D} = \{D_1, D_2, ..., D_n\}$, such that x_1 takes value in D_1, x_2 takes value in D_2 and so on, under a set of constraints $\mathcal{C} = \{C_1, C_2, ..., C_e\}$. Given a constraint $C \in \mathcal{C}$, $var(C)$ is the sequence of variables involved in the constraint (also called its *scope*, or in some cases *schema*), while $rel(C)$ is the set of value tuples on $var(C)$ which are permitted by the constraint, $rel(C) \subset \prod_{x_i \in var(C)} D_i$. Often, $rel(C)$ is not given in extension, but providing a function that determines whether a tuple is permitted or not by C. The number of variables involved in C, $|var(C)|$, is the arity of C. If the arity of C is 1, 2, ... we say that C is a unary, binary, ... constraint. A constraint network is binary if all its constraints are unary or binary.

Often it is implicitly assumed that only may exist a single constraint among a particular set of variables. Constraint networks with this property are said to be normalized. In the following, we always assume normalized constraint networks. A unary constraint on x_i is written as C_i, and a binary constraint between x_i and x_j is written as C_{ij}.

Several operations can be done on a constraint network. One of these possible tasks is to assign a value to each variable from its domain such that *all* constraints are satisfied. This task is called *solving* the network, which is also referred as the *constraint satisfaction problem* (CSP). Most of algorithms presented here consider the task of solving a constraint network; their complexities are given in terms of n –the number of variables, d –the maximum domain size– and e –the number of constraints–. This classical CSP definition assumes *hard* constraints: they are mandatory and any solution must satisfy all of them (in contrast with the *soft* constraint concept, see Section 3.5).

A particular case of CSP is the SAT problem, defined as finding a consistent assignment for a propositional formula expressed in clausal form satisfying the formula. A SAT instance can be seen as a CSP as follows: SAT variables correspond to CSP variables with domains $\{true, false\}$, and each clause corresponds to a constraint requiring that at least one literal (affirmed or negated variable) of the clause has to be assigned to $true$.

A constraint network $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ can be visualized using a graph representation. There are three popular representations:

- Primal graph: each node represents a different variable, and there is an edge between two nodes if the corresponding variables appear together in the scope of some constraint.
- Dual graph: each node represents a different constraint, and there is an edge between two nodes if the corresponding constraints share a common variable in their scopes.
- Hypergraph: each node represents a different variable and each hyperedge represents a different constraint. The hyperedge corresponding to constraint c connects

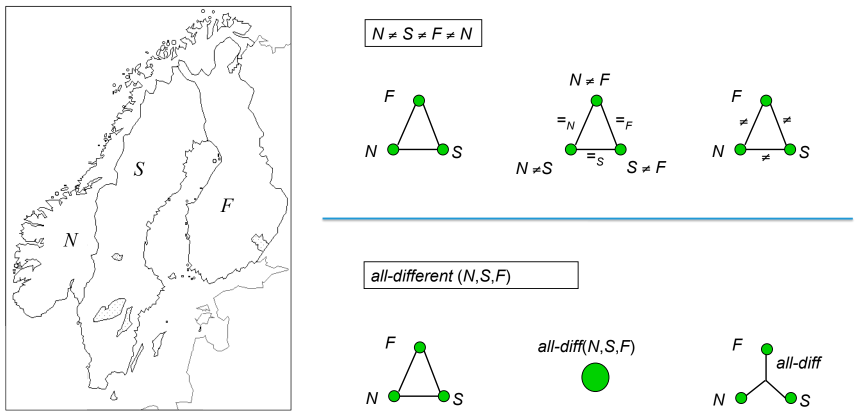

Fig. 1. Left: the map formed by Norway, Sweden and Finland as instance of the map coloring problem. Right: primal graph, dual graph and hypergraph of: (up) this instance formalized with binary constraints; (down) this instance formalized with a single *all-different* constraint.

all variables in $var(c)$. When the constraint network contains unary and binary constraints only, the hypergraph coincides with the primal graph.

If, in addition to the constraint network $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, there is an objective function on a subset of the variables, a popular task is to optimize that function subject to the constraints. This task is usually called the *constraint optimization problem* (also known by the acronym COP). About complexity, it is well-known that these problems are intractable in general: classical CSP is NP-complete (it is enough to realize that 3-SAT, a famous NP-complete problem, can be seen as a CSP), while COP is NP-hard (specifically, Δ_2-complete).

As examples, we mention two classical ones: the *map coloring problem* and the *n-queens problem*. The map coloring problem consists of assigning colors from a given set to regions in a map, such that two adjacent regions have different colors. This problem can be easily expressed as a constraint network: variables are regions, values are colors, all variables have the same domain, constraints are $\neq$ between variables corresponding to adjacent regions. The map of Norway, Sweden and Finland is a nice example of map coloring. It appears in Figure 1, with its corresponding graphs.

The n-queens problem consists of locating n queens in a $n \times n$ chessboard such that no queens are attacking each other. This problem admits several formalizations in terms of constraints. One of them considers variables representing queens: there is one variable per row (obviously, two queens cannot be located at the same row), the values are the available columns (initially, from 1 to n), under the constraints that two variables cannot be in the same column and in the same diagonal of the chessboard. This formalization appears in Figure 2.

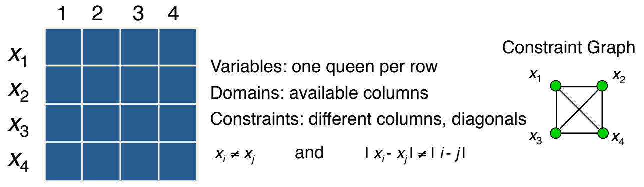
1
2
3
4
x_1
x_2
x_3
x_4
Variables: one queen per row
Domains: available columns
Constraints: different columns, diagonals
$x_i \neq x_j$ and $|x_i - x_j| \neq |i - j|$
Constraint Graph
x_1
x_2
x_3
x_4

Fig. 2. The 4-queens problem as CSP: variables represent rows and values are the available columns of the chessboard. The constraint graph is a clique of binary constraints.

3 Constraint Solving

Given a constraint network $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, the task of interest finding an assignment of values to variables such that all constraints are satisfied (= solving the network). This task is known as the *constraint satisfaction problem* (CSP). This can be done by several methods; we describe next those which we consider more relevant.

In this Section, several algorithms assume binary constraints. This is done for the easiness of presentation, nowadays constraint solving approaches have been generalized to handle constraints of any arity.

3.1 Search

Conceptually, the simplest way to solve a CSP is by searching the state space of the problem. This is done by guessing values for the problem variables. There are two main families of search methods: systematic search and local search. Both are described next.

Systematic Search. Since the number of variables is finite, and the number of values in their domains is also finite, the state space –although it may be very large– is finite, so it can be traversed systematically until exhaustion. This is the rationale of systematic search for solving constraint networks. The search space is represented as a tree, where each variable x_i is associated with a level in the tree, and each node at this tree level has d_i successors at the next level, one for each possible value of x_i ($d_i = |D_i|$). Each tree node corresponds to one assignment of variables (the one specified by the unique path from this node to the root); interior nodes correspond to partial assignments, while leaf nodes represent complete assignments. The search tree for the 4-queens problem appears in Figure 3.

The simplest algorithm one could think of follows a *generate-and-test* schema. It generates all search tree leaves (or equivalently, all complete assignments involving all variables), testing the constraints on each of them. Any assignment satisfying all constraints is a solution. This algorithm is simple but extremely inefficient because it may test a exponential number of assignments which are inconsistent due to the same reason. For instance, consider the 4-queens problem. We know that any assignment with $x_1 = 1, x_2 = 1$ is inconsistent, because the queens in the first two rows are in the same

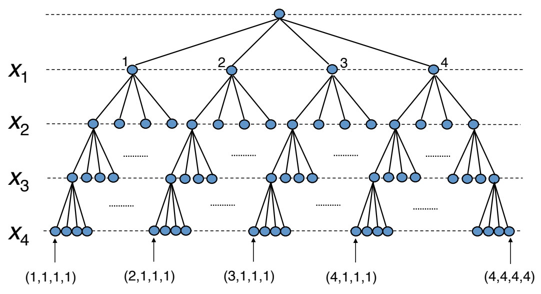

Fig. 3. Search tree of the 4-queens problem, following formalization of Figure 2.

column. However, this algorithm will generate all combinations of values for queens in the third and fourth rows, to finally conclude that there is no solution below.

The *backtracking* (BT) algorithm [48, 15] solves some of the issues of generate-and-test. It performs a depth-first search traversal of the search tree, checking at each node whether all constraints among assigned variables are satisfied or not. If the answer is yes, BT continues its depth-first traversal. If the answer is no, BT prunes the subtree below the current node and continues depth-first in the rest of the tree. The justification for pruning the subtree rooted at the current node is simple: if at that node there is a constraint among assigned variables which is violated, it will remain violated in all the nodes of the subtree below the current node. There will not be any solution below the current node, so there is no point searching such subtree. The spatial and temporal complexities of BT are those of depth-first search: exponential in time but linear in space.

BT is a chronological backtracker, that is, when it finds a dead-end it changes the last decision taken. Sometimes, this strategy is not effective because the last decision taken may not be the culprit for the dead-end. As an example, consider the instance of the 8-queens problem depicted in Figure 4. The last decision taken –the value assigned to x_5– is not the culprit for not having a consistent value for variable x_6. *Dependency-directed backtracking* [88] and the most general *intelligent backtracking* [17] solve this issue, but requiring exponential memory in its most general form. Keeping linear memory, several restricted forms of intelligent backtracking for constraint search have been developed under the names of *backjumping* [42], *graph-based-backjumping* [25], *conflict-directed-backjumping* [74] (CBJ). The idea of CBJ in the context of binary constraint networks is as follows: each variable keeps a conflict set, where for each value it records the first assigned variable which is incompatible with that value. When the search algorithm finds a dead-end, it backtracks to the closest variable in its conflict set (instead of backtracking to the closest variable, as BT will do). Upon backtracking, the conflict set of the new current variable is enlarged with the conflict set of the old current variable, minus the new current variable. Intermediate variables between the jumping variable and the arrival variable are desassigned.

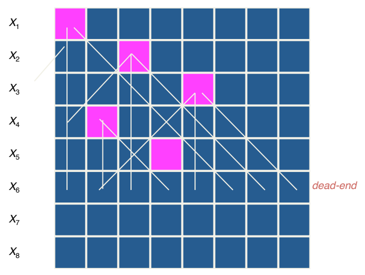

Fig. 4. A state of the 8-queens problem. There is no consistent value for x_6 but backtracking to x_5 does not release any consistent value for x_6. Intelligent backtracking jumps directly to x_4.

A further step in this line is the *dynamic backtracking* (DB) algorithm [45]. It is an intelligent backtracking algorithm which requires linear memory, while keeping some elaborated information about the culprits for constraint violations. When facing a dead-end, DB backtracks to the closest variable responsible for the conflict, keeping the values of the variables between the jumping variable and the arrival variable. This more sophisticated backtracking may have also some caveats regarding efficiency [4].

A last word for *variable and value ordering heuristics*. Associating statically one variable with each tree level, so this variable always appears at the same position when BT traverses one branch down, is more than needed to assure BT completeness. What is required is that each variable has to be considered in each branch, but the order in which the same variable appears may differ from branch to branch. This observation gives rise to *dynamic variable ordering* (also called DVO) heuristics, trying to select dynamically the variable which shortens most the remaining search time. Solving a constraint network, most of the time is spent traversing branches which do not lead to any solution; in such case, a good strategy is to detect as soon as possible when there is no solution below the current node to perform pruning and jump to another branch. This is the basis of the *first-fail* principle: select the variable that is most likely to fail [49]. A simple implementation of this principle is selecting as the next variable the one with less consistent values with previous instantiations (called *minimum remaining values* [3] or simply *min-domain* heuristic [87]; often it is presented in connection with constraint propagation). Assuming the task of finding one solution, a similar reasoning can be done on value ordering as follows. When generating the successors of a node, that is, the possible assignments for the variable associated with that node, any value ordering is legal. BT will stop after finding one solution that necessarily will contain all assigned variables. Since a value has to be chosen for the current variable, a reasonable heuristic is to start with those values which are most likely to succeed. BT starts searching successors from left, so values should be ordered left to right from most to less likely to succeed.

Local Search. While systematic search procedures keep somehow memory of where they are in the search space, in order to perform an exhaustive traversal, local search procedures only keep information local to the current state. Standard local search procedures cannot guarantee that they will find a solution if there is one, or that a particular state will not be visited more than once. While lacking these theoretical guarantees, local search procedures usually scale up better than systematic ones; because of that, local search is sometimes preferred when the size of the problem to solve is large, often in the context of constraint optimization.

There are a wide set of search strategies: evolutionary algorithms [47], simulated annealing[1] [55], tabu search [46], just to name a few. Often they are named under the umbrella term of *metaheuristics* [16]. All them consider three basic elements (here we consider states[2] s with all variables instantiated):

- Objective function $F(s)$: it maps each state s to a cost given by $F(s)$; the solution is the global minimum of such function.
- Neighborhood $N(s)$: set of states s' where you can go from the current state s in the next iteration. Typically, s' differs from s in the values of i variables (typically i is between 1 and 2).
- Selection criterion: given $s, F(s), N(s)$, how to choose the next state s'.

A local search procedure considers the greedy minimization of function $F(s)$ following an iterative approach. At each iteration, after moving to the current state s, it computes the values of $F(s)$ and $N(s)$. It moves from state s to s' if $F(s) \geq F(s')$, starting a new iteration. However, this procedure may get trapped in local minima, where $F(s') \geq F(s), \forall s' \in N(s)$. In some cases, this issue is tackled with restarts or random moves (since in the next state $F(s')$ may be worse that $F(s)$, these moves are called *worsening moves*). This combination is named *stochastic local search* (SLS) [52]. A SLS procedure ends when it finds a solution or when exhausting computational resources (such as a maximum number of iterations). There is a large number of combinations of SLS methods for CSP solving. Next, we summarize some of the most successful ideas in SLS:

- Min Conflicts Heuristic [68]: $F(s)$ counts the number of not satisfied constraints (conflicts) in s; the next state s' tries to minimize the number of violated constraints, while changing the value of one variable only.
- Penalty Methods [72, 24, 92]: each constraint has a weight and $F(s)$ includes the sum of weights of violated constraints; weights are not constant but they change during search (for instance, in [72] when the system reaches a local minimum, the weights of violated constraints are incremented).
- GSAT [83]: this is a procedure to solve SAT instances, a particular case of CSP. Given a boolean formula in clausal form and a complete variable assignment (chosen initially at random), at each step GSAT chooses a boolean variable and flips its

[1] In some communities, simulated annealing is considered as a global search strategy. Here it is included into the local search class as a non-systematic search method.

[2] Following the concept of state space in Artificial Intelligence, a *state* is a particular configuration of variables; in some cases, some variables might not be assigned (as in the interior nodes of the search tree of Figure 3).

value (from *true* to *false* or vice versa) such that the number of unsatisfied clauses is minimized. If a solution has not been found, GSAT restarts every *maxSteps* flips. The whole process iterates a maximum of *maxTries*.

- Random Walk [84, 65]: instead of performing worsening moves only when the system is stuck in a local minimum, a worsening move can also occur in non-local minima with a small probability.
- Large Neigborhood Search [18]: when computing $N(s)$ some parts of the current state s (assumed feasible) are maintained while others are varied. Then, $N(s)$ includes states that differ in more than one or two values from s. Using consistency techniques, $N(s)$ is pruned, keeping its size manageable.

Local search appears as a promising strategy for solving constraint-based problems [51].

3.2 Inference

Processing a constraint network by *inference* means to perform some legal operations on constraints, such that original constraints are modified and a new constraint network naturally replaces the original one. The new network has exactly the same solutions as the original one. The interesting point is that the new constraint network is more explicit than the original one, and this allows traversing its search space more efficiently when looking for a solution. In this sense, we say that new network is *easier to solve* than the original.

Inference methods are divided in two large families: incomplete and complete inference. The main difference between them is that complete inference methods produce directly the solution after their execution, while this is not true for incomplete inference methods. The most relevant methods of both families are described next.

Incomplete Inference. *Incomplete inference* is a umbrella term for a number of procedures working on constraints such that, when applied on any constraint network, although reducing the search space of the original network they do not produce directly the solution, which has to be found by subsequent search. Procedures manipulating and simplifying constrains have been developed since the very beginning of Constraint Reasoning,[3] first as preprocessing and later combined with search.

Incomplete inference is also known as local consistency. An essential part of it is constraint propagation, when changes in a part of the problem (such as assigning a variable or removing values in a domain) are propagated to other parts of the problem because the action of constraints (typically, by enforcing some necessary condition for solutions). When the effect of local consistency is just removing values from domains, it is also called domain filtering.

ARC CONSISTENCY. A form of incomplete inference is arc consistency. Basically, it consists of analyzing each constraint in turn, to detect elementary values of variables that are not permitted by the constraint. These values are removed from its domain

[3] The early work of Waltz in vision [95] considers arc consistency on line edges.

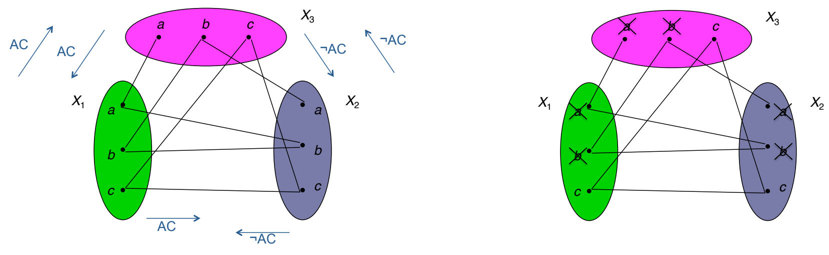

Fig. 5. Example of arc consistency processing. Left: clique of three variables, connected by binary constraints; permitted value pairs are connected by a link (*microstructure*). Right: crossed values are eliminated from their corresponding domains because they are arc inconsistent; the problem solution is the assignment of value c to each variable.

(because they will not be in any solution), and their removal is propagated through other constraints. This process continues until finding that no further changes are induced in the network. Then, we say that we have reached a *fix point*.

Let us consider a binary constraint network, where constraints correspond to arcs in the primal constraint graph (C_{ij} connects x_i and x_j). Instead of considering the whole constraint network, we consider each subnetwork of 2 variables connected by a constraint. Analyzing such constraint it is possible to detect if some values will not be in a solution. Then, these values are removed and the effect of their removal is propagated to other domains by the rest of constraints. If a variable becomes without values (empty domain), the network has no solution; in this case, the ”no-solution” condition has been found inspecting subnetworks of size 2 only. Otherwise, if some values have been removed, the search space of the instance becomes smaller and search can be done more efficiently.

More formally, constraint C_{ij} is directionally arc-consistent (from x_i to x_j) iff for any value $a \in D_i$ there exists a value $b \in D_j$ such that $(a,b) \in rel(C_{ij})$. C_{ij} is arc-consistent iff it is directionally arc-consistent in both directions. A constraint network is arc-consistent iff all its constraints are arc-consistent [28].

If for a particular $a' \in D_i$ there is no value in D_j consistent with it, a' can be removed from D_i because it will not be in any solution (the assignment $x_i = a'$ will violate at least one constraint, C_{ij}). The removal of a' is propagated by all other constraints involving x_i. Let us consider C_{ik}. If it was previously made arc-consistent, a' removal may cause C_{ik} to become directionally arc-inconsistent, from x_k to x_i, while in the other direction it remains arc-consistent. So C_{ik} has to be rechecked from x_k to x_i. An example of arc-consistency processing appears in Figure 5.

A number of algorithms have been proposed for enforcing arc consistency on a binary constraint network. Among them, we mention AC-3 [62], the optimal AC-4 [70] and AC-2001 [13]. Arc-consistency can be achieved in a polynomial number of steps (optimal temporal complexity $O(ed^2)$, e is the number of binary constraints, d is the largest domain size).

Originally, arc consistency was developed for binary constraint networks, when the hypergraph coincides with the primal graph. When considering constraints of any arity, the same concept is called *generalized arc consistency* (GAC). A constraint C is GAC iff for any $x_p \in var(C)$, a value $a \in D_p$ appears in some permitted value tuple $t \in rel(C)$ as value for x_p. A constraint network is GAC if all its constraints are GAC [28].

Less demanding than arc consistency is *bound* consistency (BC), applicable to totally ordered domains. Only the lower/upper bounds of each domain are checked in each constraint using a weaker form of consistency that assumes the domains of other variables compact and without holes, perfectly defined by their lower/upper bounds (there are some variants on the definition of bound consistency, see [9]).

Arc consistency is one of the topics on which more work has been done in the context of constraint processing. For a comprehensive study, see [9].

PATH-CONSISTENCY. Assuming again binary constraint networks, let us consider subnetworks of three variables, say x_i, x_j, x_k. If a pair of consistent values between x_i and x_j cannot be consistently extended including x_k, the initial pair is considered inconsistent and recorded in a modified C_{ij}. If at some point a constraint becomes empty (= no pair of values is permitted), the network has no solution.

Formally, a pair of variables (x_i, x_j) is path-consistent iff for any pair of consistent values $(a, b) \in rel(C_{ij})$ and any third variable x_k, there exists $c \in D_k$ such that $(a, c) \in rel(C_{ik})$ and $(b, c) \in rel(C_{jk})$. A constraint network is path consistent iff any pair of variables is path-consistent [28].

As in the arc consistency case, if constraints are given in extension path consistency can be achieved in polynomial time (the optimal complexity is $O(n^3d^3)$ [70]). However, the complexity is substantially higher than in the arc-consistency case, so usually it does not pay-off when compared with the simpler but cheaper arc-consistency property (especially when local consistencies are enforced at each node visited by a search procedure, as described in Section 3.3). In addition, path-consistency modifies the constraints, which is more difficult to handle than the easier task of modifying the domains, the only effect of arc consistency. Because of all these, path consistency remains unattractive in practice, and it is usually not available in most existing constraint solvers.

K-CONSISTENCY. K-consistency is a generalization of the above concepts. A binary constraint network is *k-consistent* iff given a subset of $k-1$ variables consistently assigned, for any kth variable it is possible to find a value consistent with the values of the $k-1$ variables previously assigned [36]. Then, arc consistency is 2-consistency, path-consistency is 3-consistency, etc. Optimal k-consistency can be achieved in time exponential in k [22]. A constraint network is *k-strong consistent*, when it is p-consistent for $1 \leq p \leq k$. An interesting result is the following: given a constraint network where variables are ordered such that the width of the order (= the maximum number of edges that go back from a node in the primal constraint graph) is lower than k, a constraint network that is strong k-consistent is backtrack-free, that is, BT will find a solution without performing backtracking [37].

This result tempts us to compute the width of a graph (and its associated ordering), processing the network afterwards to achieve the required level of strong consistency. However, the procedure for strong consistency adds edges to the primal graph, so the width of the selected ordering may change after executing this procedure (except when the constraint graph is a tree, then it can be solved backtrack-free after processing it by directional arc-consistency).

Interestingly, the *adaptive consistency* approach [29] was a step forward along this line. First, it was able to adjust the level of consistency required for each variable of the ordered primal graph to assure a backtrack-free search. And second, it introduced the concept of induced width, as the width of the ordered graph after adaptive consistency processing. But all this is better explained as bucket elimination in the next paragraphs.

Complete Inference. *Complete inference* procedures, when applied to any constraint network, produce directly a solution so no subsequent backtrack search is needed. Let us assume that all constraints are given in extension. A *tuple* t with scope S is an ordered set of values assigned to each variable in $S \subseteq \mathcal{X}$. The *projection* $t[T]$ of a tuple t on $T \subset S$ is a new tuple which only includes the values assigned to the variables in T. We define the join $\bowtie$ of two constraints C and C', written $C \bowtie C'$ as a new constraint with scope $var(C) \cup var(C')$ such that $t \in rel(C \bowtie C')$ if and only if $t[var(C)] \in rel(C)) \wedge t[var(C')] \in rel(C')$. The join of constraints is associative and commutative.

By definition, we observe that the tuples satisfying constraints C and C' satisfy $C \bowtie C'$, and vice versa. Then, a direct way to compute all solutions of a constraint network is computing the join of all its original constraints, $\bowtie_{C \in \mathcal{C}} c$. Tuples satisfying $\bowtie_{C \in \mathcal{C}} C$ are the solutions of the original network, and there are no other solutions.

Since solving a constraint network is NP-complete, it causes no surprise that complete inference has temporal complexity exponential in the *induced width*, w^*, a parameter of the primal constraint graph that basically measures its cyclicity.

BUCKET ELIMINATION. As mentioned before, the join of all the original constraints will produce a single constraint whose permitted tuples will be the solutions of the problem. However, this is more than needed to solve the problem by inference. If we do the joins of constraints following some particular order, we can save space and time. To properly describe the bucket elimination algorithm, we still need to define a new operation: given a constraint C, projecting a variable $x \in var(C)$ out of C, written $C[-x]$, is a new constraint with scope $var(C) - \{x\}$ whose allowed tuples are those of C eliminating the values of x and removing duplicates.

Before describing the bucket elimination algorithm [27], we introduce some concepts from [37]. A *ordered graph* (G, o) is simply a graph G with a total order o of its nodes. Given (G, o), the *width of a node* x is the number of arcs that connect x with other nodes prior x in the order o; the *width of an order* o is the maximum width of its nodes; the *width of a graph* is the minimum width of all its possible orderings. Given (G, o), where G is a primal constraint graph, we define the *induced graph* G' along the ordering o [29] as G enlarged with some extra arcs computed as follows: nodes are processed from last to first along o; let x a node, all its parents (nodes connected with x and before x in o) are connected by a clique of arcs. Given the induced graph G', previous definitions of width associated to node, order or graph also apply here. In particular, the parameter called induced width $w*$ is the width of the induced graph G'. Computing the minimum induced width of a graph is NP-complete [28].

The bucket elimination algorithm works as follows: given an ordered graph (G, o) of a constraint network, the bucket of each variable is the set of constraints which mention that variable as the last variable in their scope following o. Buckets are processed following o from last to first. If x is the current variable, all constraints in bucket B_x are combined together into a single constraint C_x which summarizes the effect of B_x. Then, since variable x is not further required, it is eliminated from C_x, obtaining the constraint R_x which replaces B_x in the set of constraints $\mathcal{C}$. This new constraint R_x –which does not mention x– is properly located in the bucket of the last variable of its scope, and it will be processed in the future.

Once all buckets have been processed, one knows if the original problem has solutions. To recover solutions, variables are processed along o –from first to last– assigning variable x a value of a tuple allowed by c_x and consistent with all previous variables in o. It is easy to see that such value always exists, and the process is backtrack-free. The spatial and temporal complexities of bucket elimination are exponential in the induced width of the constraint graph [27].

MINI-BUCKETS. It may happen that the system has not enough memory to compute $C_i \longleftarrow\bowtie_{C \in B_i} C$ (or you want to be faster than computing this exponential-time step). To alleviate these exponential bounds, the *mini-bucket* approximation comes into play. The idea is to partition bucket B_i in subsets such that the join of all constraints in each subset produces a new constraint whose arity is upper bounded by r, a parameter that establishes the maximum memory available to keep the new constraint (at most d^r tuples) [26]. Mini-bucket elimination is an approximation of the exact bucket elimination algorithm, it is not guaranteed to find a true solution (although the final assignment will be closer to a solution). The spatial and temporal complexities of mini-bucket elimination are exponential in r. Mini-buckets can also be used to generate heuristics for optimization tasks [54].

It is possible to use search as an alternative to the mini-buckets approximation. Given an ordering o of variables to be processed by bucket elimination, and r an upper bound on the arity of the new constraints, this approach works as follows: if processing the bucket of a variable along o produces a new constraint of arity $\leq r$, then the standard bucket elimination algorithm is used; otherwise, a search algorithm is used, branching on the possible values for this variable [59].

3.3 Hybrid Algorithms

Incomplete inference (also called local consistency) procedures can be used either (i) as preprocessing, before search starts, to simplify the original instance, or (ii) inside the search process, typically at each visited node. In the second case, we talk about *hybrid algorithms* because they combine search (typically systematic search, in its simplest form the BT algorithm) with some form of incomplete inference. This combination often offers significant savings if compared with pure search, and it has become a kind of standard when solving constraint networks. The benefits of this combination are due to the reduction that local consistency causes in the search space of subproblems found by search, causing to visit less nodes. On the other hand, some extra work is done when enforcing the local consistency at each visited node (because of that, local

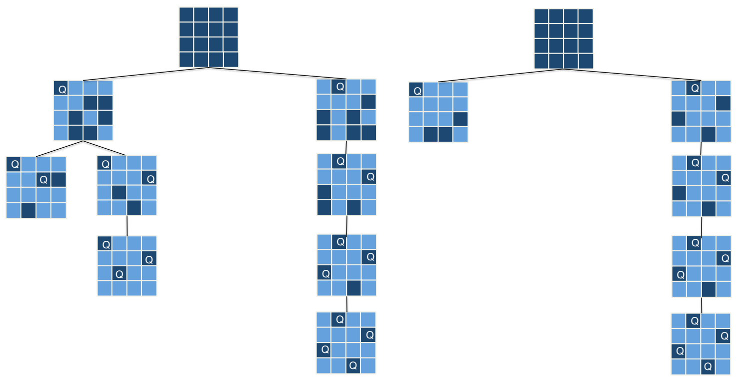

Fig. 6. FC search tree (left) and MAC search tree (right) on the 4-queens problem. Observe the more powerful MAC inference in the left part of the search tree. Light cells are forbidden, dark cells are free.

consistencies of low complexity are preferred). It has been observed experimentally that the best trade-off between benefits and extra work occurs for arc consistency (or lower local consistencies). Because of that, we will concentrate our presentation on hybrid algorithms to the combination of BT with some forms of arc consistency.

Although hybrid algorithms often offer better performance, their worst case temporal complexity does not improve over the BT one, which is exponential. An interesting work compares the relative performance of some hybrid algorithms [57].

Forward Checking. Originally proposed for solving binary constraint networks, the *forward checking* (FC) algorithm [49], follows a BT schema with the following addition: when a variable is assigned, constraints connecting this variable with unassigned ones are made arc-consistent. This may cause deletions in the domains of these variables; if one of these domains becomes empty, there is no solution below the current node so the subtree below it can be pruned, saving search effort. Upon backtracking, when the value assigned to a variable changes, the deletions caused by enforcing arc consistency with the old value for that variable have to be undone. The search tree developed by FC in the 4-queens problem appears in Figure 6.

In FC, it is worth noting that when a variable is assigned a value, no checking for consistency is done on already assigned variables because all values remaining in the domains of unassigned variables are already consistent (because of FC behavior). The FC algorithm has been extended to deal with non-binary constraints in [12].

Maintaining Arc Consistency. The *Maintaining Arc Consistency* (MAC) algorithm [79] represents an step further in the application of arc consistency inside a systematic search schema. The main idea of this MAC is that arc consistency must be maintained at all times in the current subproblem considering all its constraints (not only constraints

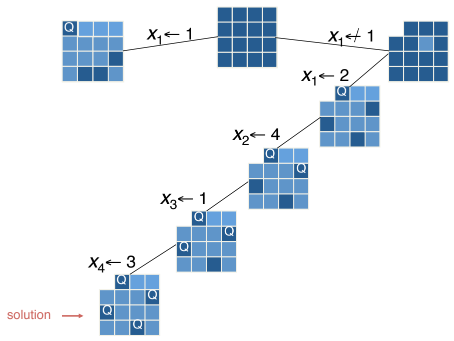

Fig. 7. MAC binary tree on the 4-queens problem. Left branches are assignments, right branches are value rebuttals (only rebuttal $x_1 \not\leftarrow 1$ appears). Light cells are forbidden, dark cells are free.

connecting the current variable with unassigned ones, as FC). In addition, the subproblem where the current variable is still unassigned must be arc consistent. This is done as follows. Let x be the current variable to be assigned. When this variable takes value a, arc consistency is enforced in the remaining subproblem. If some domain becomes empty there is no solution below the current node (observe that this step is stronger than the corresponding step in FC, because arc consistency is enforced on more constraints), so the subtree below it can be pruned causing search savings. Otherwise, a recursive call is done indicating which is the next variable. If no solution can be found under the assignment $x \leftarrow a$, then a is removed from D_x and arc consistency is enforced on the new subproblem. This step is new with respect to FC. As in the previous algorithm, upon backtracking when the value assigned to a variable changes, the deletions caused by enforcing arc consistency with the old value for that variable have to be undone. The search tree developed by MAC in the 4-queens problem appears in Figure 6.

An original view of the MAC algorithm is that of traversing a binary tree where left branches correspond to assignments of variables ($x = a$) and right branches correspond to rebuttal of the same values (adding the constraint $x \neq a$). An example of such binary tree on the 4-queens problem appears in Figure 7.

3.4 Algorithmic Enhancements

Symmetry Exploitation In many constraint networks there are symmetries: transformations involving variables and values that leave the problem invariant. For instance, in the 4-queens problem following Figure 2 formulation, if we exchange x_1 with x_4 and x_2 with x_3, we obtain exactly the same problem (this is called *variable symmetry*). Alternatively, if we exchange columns 1 and 4, and columns 2 and 3, again the problem remains the same (this is called *value symmetry*). The importance of symmetry comes from this simple observation: if a state is solution, all its symmetric states are also solutions (and the same occurs with non-solutions). Therefore, it is enough to visit one state of all states related by a symmetry (ignoring the symmetry these states look different, but considering the symmetry these states belong to the same equivalence class

–equivalence relation induced by the symmetry– and it is enough to visit one representative of such class). In some cases, this causes enormous reductions in the size of the space to explore, producing very substantial gains in search efficiency. Because of that, symmetries have received some attention in constraint solving.

Variable and value symmetries have been known for long, but a general definition of symmetry in Constraint Reasoning was provided not so long. In [21], authors propose (1) *solution symmetry*: a permutation of variables and values that leaves the set of solutions invariant, (2) *constraint symmetry*: a permutation of variables and values that leaves the set of constraints invariant (so basically the problem does not change). Both types of symmetry allow variable/value symmetries as special cases. Interestingly, constraint symmetry is included in solution symmetry; in fact, there are many more solution symmetries than constraint symmetries. Programmers detect usually constraint symmetries only because without the set of solutions it is difficult to identify solution symmetries that are not constraint symmetries [21].

Perhaps the simplest way to deal with symmetries is by model reformulation, looking for a model with a lower number of symmetries. Unfortunately, this technique is really case-dependent, and requires a deep knowledge of the problem at hand (see [43] for some examples). When the problem presents a number of indistinguishable variables, typically we numbered them, introducing artificially symmetries that do not exist in the original problem. A way to solve this is to use set variables (whose value is a set instead of a single value). Set variables break some variable symmetries and represent more naturally set solutions.

An approach to deal with symmetries is to add some constraints before search starts to break symmetries. For instance, imagine that your problem has ten variables which are fully symmetric, $x_1, x_2, ...x_{10}$ (for any pair of variables, you can freely exchange them). A way to deal with this symmetry is to add constraints requiring a lexicographical order among them: $x_1 \leq x_2 \leq x_3 \leq ... \leq x_{10}$. This approach is called *lex-leader* [23]. This strategy requires a static variable ordering compatible with the lex-leader ordering. In some cases, a matrix of variables appears in constraint programming; if rows are interchangeable and columns are interchangeable, lex-leader constraints can be imposed in rows and columns, producing what is called *double-lex* approach [56]. Also lexicographic ordering over multisets has been investigated.

Alternatively, one can break symmetries by adding some constraints dynamically during search. One approach following this line is SBDS (Symmetry Breaking During Search) [44]. The basic idea is, upon backtracking, to add constraints that prevent to visit states that are symmetrical to those already visited. Another approach –somehow related– is called SBDD (Symmetry Breaking by Dominance Detection) [33]. The basic idea is, when going to visit a new node of the search tree, check if this node is symmetrically equivalent to an already visited one. To do so, it is not needed to store all visited nodes; it is enough to store the roots of fully explored subtrees. Unlike adding symmetry breaking constraints before search, both SBDS and SBDD can be used with dynamic variable ordering heuristics and they can deal with symmetries involving values.

In some cases, the number of symmetries could be large (even exponential). Adding static or dynamically new constraints to break these symmetries could be counterproductive at some point. Checking extra constraints causes some overhead, to be com-

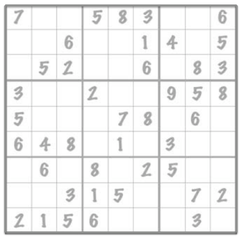

Fig. 8. The Sudoku completion problem: assign a digit from 1,...,9 to each empty cell such that the assignment of the whole grid forms a legal Sudoku (as explained in the text).

pared with the benefits of symmetry breaking. In practice, it is commonly assumed the best efficiency is achieved breaking a only part of the symmetries present in the problem. For a more in-depth presentation, see [43].

Global Constraints. MOTIVATING EXAMPLE. A good way to introduce global constraints is with an example. Let us consider the Sudoku puzzle: it is a grid of 9×9 cells, which contains 9 non-intersecting subgrids of size 3×3. The goal is to assign to each cell a digit from 1 to 9, such that the following constraints are satisfied:

- there is no repeated digit in each row of the 9×9 grid,
- there is no repeated digit in each column of the 9×9 grid,
- there is no repeated digit in each 3×3 subgrid.

While finding a legal assignment for an empty Sudoku is trivial, the problem usually considered is the Sudoku completion: when some cells have already a digit assigned and the task is to find an assignment of digits to the other cells such that the whole assignment forms a legal Sudoku. An example of Sudoku completion appears in Figure 8.

A natural way to model this problem as a CSP is to consider each cell as a variable. Empty cells have a domain $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, while cells with a digit assigned have a domain formed by that digit, under the above mentioned constraints. It is not difficult to see that Sudoku constraints can be expressed as cliques of binary inequality constraints among the variables involved. Thus, there are 36 binary inequalities per row, 36 binary inequalities per column and 36 binary inequalities per 3×3 subgrid.

Alternatively, we can consider the *all-different* constraint on a set of variables `SetVar`, requiring all variables of `SetVar` to be assigned a different value. Using this constraint, we can model the Sudoku completion problem with one *all-different* constraint per row, one per column and one per 3×3 subgrid.

It is clear that, when a variable x is assigned a value v, that value has to be removed from the domains of the other variables constrained with x. In addition, the number of different available values (cardinality of the union of the domains) for the 9 constrained variables (corresponding to the same row, column or 3×3 subgrid) should be 9; otherwise, there is no way to satisfy the corresponding Sudoku constraint. The first reasoning is achieved when enforcing GAC in both formalizations, clique of binary inequalities and *all-different* constraints. The second reasoning is only achieved when enforcing GAC on *all-different* constraints; a binary inequality does not include all the variables involved in a Sudoku constraint, so enforcing GAC on it cannot account for the number of their different available values.

Why *all-different* has more pruning power than the clique of binary constraints? The reason is based on the set of variables viewed by each constraint. The *all-different* constraint considers 9 variables simultaneously, so it is able to make better deductions than when considering pairs of variables in isolation. Having a more *global* view of the problem allows for better deductions, which cannot be achieved with a more local view (which in occasions could be myopic). An example of this appears in Figure 9: a map coloring instance with 3 variables constrained to have different values but only two available values.

It is worth noting that both formalizations for the Sudoku completion problem are correct and obtain true solutions. However, assuming a solving strategy based on search, one is more efficient than the other because it is able to detect (and therefore prune) sooner than the other branches not leading to any solution.

GLOBAL CONSTRAINTS. A *global constraint* captures the relation among a non-fixed number of variables with specific semantics [90]. C is a class of constraints defined by a Boolean function f_C whose arity is not fixed. Constraints $C(x_{i_1}, \ldots, x_{i_k})$, $C(x_{j_1}, \ldots, x_{j_q})$ with different arities can be defined by the same Boolean function. For instance, *alldifferent*(x_1, x_2, x_3) and *alldifferent* (x_4, x_5, x_6, x_7) are two instances of the *alldifferent* global constraint, where $f_{\textit{alldifferent}}(T)$ returns true iff $x_i \neq x_j$, $\forall x_i, x_j \in T$.

Examples of global constraints are:

- *alldifferent*: different values have to be assigned to a set of variables;
- *atmost/atleast*: an upper/lower bound on the number of times a value may appear assigned in a set of variables;
- *cardinality* (also written *gcc*): it establishes lower and upper bounds to the number of times that several values may appear assigned in a set of variables;
- *sum*: the addition of the numerical values of a set of variables should be equal to some parameter;
- *circuit*: it is satisfied when a set of variables are assigned nodes of a directed graph such that they form a Hamiltonian circuit;
- *lexicographic*: considering two vectors of variables, it is satisfied when the two vectors are lexicographically ordered after assignment.

A global constraint is defined by a boolean function, no matter the arity of the constraint. This permits to develop GAC enforcing programs for this particular global constraint in which the number of variables involved in the constraint is an input parameter. Exploiting the semantic of this global constraint, those GAC enforcing programs can be made specific for it, with lower complexity than the corresponding to a generic GAC propagator. As an example, consider the *alldifferent* constraint on k variables; a generic GAC propagator requires to pass through all the tuples so it has a complexity $O(d^k)$, while the specialized GAC propagator [77] can do it in $O(k^2 d^2)$. The exploitation of

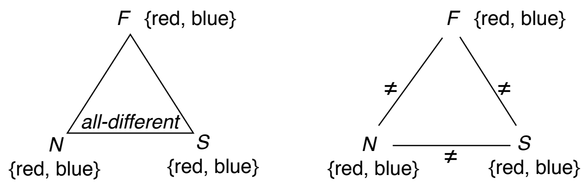

Fig. 9. Map coloring instance, with three variables but only two available colors formulated with an *alldifferent* constraint (left) and its binary decomposition into a clique of binary inequalities (right). While GAC on the binary decomposition deduces nothing, GAC on *alldifferent* detects that there is no solution.

this fact has been crucial for the success of the constraint technology in practice (see Section 5).

Some global constraints are *semantically decomposable* in a polynomial number of simpler constraints of fixed arity, in the sense that the set of solutions of the global constraint is exactly equal to the set of solutions of a polynomial number of simpler constraints on the same set of variables [10]. An example of this is the *alldifferent* constraint: *alldifferent*(x_1, x_2, x_3) is semantically equivalent to $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$. However, there are constraints that cannot be decomposed into simpler constraints without adding a polynomial number of extra variables. An example of this is the *atleast* constraint. Allowing extra variables, any constraint is decomposable in smaller constraints.

A global constraint is *operationally decomposable* into smaller constraints of fixed arity, if enforcing GAC on the global constraint removes the same values as enforcing GAC on the set of smaller constraints. A global constraint may be semantically decomposable into a set of smaller constraints but not operationally decomposable in such set, in the sense that GAC on the global constraint is more powerful (removes more values) than GAC on the decomposition. For example, *alldifferent* is decomposable in a clique of binary inequalities; however, GAC on *alldifferent* prunes more than binary AC on the set of inequalities (see Figure 9). So it is interesting to find decompositions which are as powerful in GAC propagation as the initial global constraint. Some constraints can do that on the same set of variables as the original constraint (for example the *lex* constraint). Others require a polynomial number of constraints on extra variables (for instance, the *atleast* constraint). And others cannot achieve the same propagation in any polynomial number of simpler constraints of fixed arity (for example, the *alldifferent* constraint) [11].

Regarding GAC propagation, there are several classes of global constraints:

- those for which GAC propagation is NP-hard (the *sum* constraint),
- those for which GAC propagation is polynomial and
  - there is no equivalent decomposition, even allowing a polynomial number of extra variables (the *alldifferent* constraint),

- there is an equivalent decomposition (the *lex* constraint on the same variables, the *atleast* constraint adding a polynomial number of extra variables).

The interest in decomposing global constraints comes from the fact that currently there are hundreds of reported global constraints (the current version of the global constraints catalog collects more than 350 global constraints, with a description of more than 2,800 pages [6]). In such situation, it is of great interest finding if there is a short set of basic constraints such that any global constraint could be seen as a combination of such basic constraints (this would facilitate enormously the implementation of CP solvers). Although some successful work in this direction has been done, some global constraints escape the intended decomposition on basic constraints.

3.5 Extensions

Continuous Constraints. The defined model of constraint network is extended to continuous domains when variable domains are intervals over the real numbers, and constraints are equations on these variables. Again, a solution is an assignment of values to variables satisfying the constraints.

This topic falls in the numerical analysis field. A common approach is to replace numerical analysis by interval analysis, where floating-point numbers are replaced by intervals. As in the discrete case, it is of great interest to discard parts of the search space that will not contain any solution. The concept of arc consistency from discrete domains has been successfully extended to the continuous case [34]. Given the computational difficulties to compute arc consistent intervals, a simpler form of local consistency called *hull* consistency (the version of bound consistency for continuous domains) is pursued. In fact, an approximation called *box* consistency, is achieved in practice [7].

Solving algorithms usually follow a branch-and-reduce schema. The idea is to obtain a set of interval boxes (Cartesian products of variable intervals) covering all solutions of the problem. The algorithm chooses (often using heuristics) a box and splits it, following a branching procedure. Resulting boxes are reduced by enforcing some kind of local consistency. The process iterates until boxes are small enough [8].

In general, solutions are found applying numerical methods (Newton method, etc.). The role of constraints is more focused to remove parts of the space without solutions, so numerical methods could be applied more efficiently. Hybrid approaches have been proposed, trying to combine different strategies. For a comprehensive description see [8].

Soft Constraints. A constraint is *soft* if an assignment violating it can still be considered a possible solution. Classical constraints are *hard*: they must necessarily be satisfied by any solution. Typically, hard constraints represent physical properties that must necessarily be satisfied, while soft constraints represent priorities or preferences, which might be violated without invalidating a solution.

There are different frameworks for dealing with soft constraints. To unify their representation, two generic frameworks have been proposed: valued CSP [80] and semiring-based CSP [14], the latter being more general than the former (essentially, they differ in the required ordering for the intermediate satisfaction values: total order for valued CSP, partial order for semiring-based CSP).

Assuming the weighted CSP model where constraints are replaced by cost functions (they return a cost that one has to pay if the instantiating tuple is in the solution), the goal is to find the assignment that minimizes the addition of costs. Solving a weighted CSP becomes an optimization problem, typically solved by branch-and-bound methods. Different lower bounds have been proposed for branch-and-bound pruning [67].

Several soft local consistencies have been proposed. Interestingly, the arc consistency in the classical hard case splits in several levels in the soft case [60]. Maintaining soft local consistency during branch-and-bound search allows to compute a lower bound. Some soft versions of global hard constraints have also been successfully used for solving soft CSP instances [91]. Inference methods in CSP solving (such us bucket elimination, or cluster-tree elimination) can also be successfully used for computing the global optimum/assignment of a weighted CSP [28].

Distributed Constraints. A CSP is distributed (DisCSP) if its parts (variables and constraints) are distributed among different agents and cannot be grouped into a single agent, often because of privacy reasons. In this setting, each agent knows a part of the problem, but no agent knows the whole problem. The solution satisfying all constraints is reached through message passing.

The first complete algorithm for DisCSP solving was Asynchronous Backtracking ABT [97]. It is a distributed search algorithm, able to perform when the problem has no solution; when a solution is found the network remains silent, which is detected by external specialized algorithms. Initially designed with a static variable ordering, ABT enhanced with dynamic variable ordering has been proposed [98]. Among other approaches we mention the AAS and AFC algorithms [85, 66]. In the context of local search (complete but with exponential memory), there is the Asynchronous Weak Commitment algorithm [97].

Replacing constraints by cost functions (as in the weighted CSP model), the so called Distributed Constraint Optimization Problem (DCOP) has received a lot of attention lately. Now the solution minimizes the addition of costs. The first complete algorithm for optimal DCOP solving was ADOPT [69], a distributed algorithm following a best-first strategy. Replacing best-first by depth-first with branch-and-bound, we obtain the BnB-ADOPT [96] which reaches better performance. These are distributed search algorithms, exchanging short but many messages (in the worst case, exponentially many). Alternatively, the DPOP algorithm [73] performs distributed inference, requiring a linear number of messages but they can be exponentially large.

4 Constraint Modelling

Modelling is the process of passing from a real-world problem into a constraint network. In principle, it is not very difficult to develop a model of a real-world problem. What could be difficult is to produce a model that solves the problem in a reasonable time. There is wide evidence that the modelling process has a dramatic effect in how easy the problem can be solved in practice [39, 76]. Despite some high level languages for constraint specification, modelling resists automatization and remains an art (sometimes a *black* art) in which constraint programmers spend their time and energy looking for efficient models of real-world problems.

Here practical experience is a grade. In a report on learned lessons on crosswords [5], authors indicate how the model, the search algorithm and the heuristics interact, and none of these decisions can be made totally independent of the others. Nevertheless, in the following we will assume that the solution is found by a combination of search and incomplete inference (interleaving search with constraint propagation is the standard solving approach), ignoring particular algorithms and heuristics (otherwise it would be very difficult to deal with all the possible combinations of these elements).

Perhaps a way to start in when assessing two constraint models is to answer two questions (i) what is the size of the search space? and (ii) which model prunes more? Let us consider the three different models of the n-queens problem:

1. One boolean variable per cell: n^2 variables, equal domains $\{0, 1\}$, constraints: different rows, columns, diagonals;
2. One variable per row: n variables, equal domains $\{1, ..., n\}$, constraints: different columns, diagonals,
3. One variable per row + *alldifferent*: n variables, equal domains $\{1, ..., n\}$, constraints: one *alldifferent*$(x_1, ..., x_n)$, different diagonals.

The search space size is: (1) 2^{n^2} (2) and (3) n^n, which is far smaller than (1). Considering constraints, model (2) includes by construction the constraint ”no two queens in the same row”, so it is preferred to model (1). Model (3) captures in a single global constraint the constraint ”no two queens in the same column”, which prunes more than the clique of binary inequalities when enforcing GAC, so (3) is preferred to (2). Therefore, simple modelling criteria are preferring smaller search spaces and constraints with higher pruning power. As a rule of thumb, we prefer models using as few constraints as possible, as long as these constraints have efficient propagation algorithms.

Existing CP solvers maintain a different level of local consistency in each constraints, looking for a reasonable trade-off between the cost of enforcing such local consistency and the search reduction it produces. Typically, AC is enforced on binary constraints, GAC is enforced on global constraints and BC is enforced on arithmetic constraints, although some variants exist. Complex constraints may have little propagation (details depend on the particular solver used).

The first election in a model is what is called a *viewpoint* [61], that basically is selecting the set of variables and their domains. From this selection, constraints must assure that model solutions correspond to problem solutions. We prefer those viewpoints which allow for an easy and concise way to express the required constraints.

Given two constraints C_1 and C_2 to be satisfied simultaneously, a new constraint can be made as the *conjunction* of these constraints $C_1 \wedge C_2$. It will be tighter than any individual constraint of the conjunction. It may cause more pruning (however, this would not necessarily speed up solving in current solvers). An example of this effect appears in Figure 10, where value 3 for x_5 would be deleted after AC if between any two queens there is one constraint ”different columns *and* different diagonals” but it will remain if between any two queens there are two constraints ”different columns” and ”different diagonals”.

The use of *global constraints* has been shown very useful in practice. On one hand, global constraints have a more global view of the problem, so typically they can filter more (= remove more values) than their binary decompositions (when such decomposition are possible) when GAC is enforced on them. On the other, exploiting their semantics, global constraint propagators can achieve GAC more efficiently than using generic propagators. Current CP solvers provide a library of global constraints, which permit programmers to use them without implementing from scratch.

A constraint is *reified* when there is a 0/1 variable associated with that constraint such that the variable takes the value 1 when the constraint is satisfied and value 0 otherwise. Some CP solvers offer reified constraints, which can be used to model disjunctions: if x_1 and x_2 are associated with constraints C_1 and C_2, the expression $x_1+x_2 \geq 1$ indicated that at least one constraint has to be satisfied.

Having the minimum set of constrains that allows for correct solutions is not always correlated with good performance. *Redundant* constraints, in the sense that they are implied from the basic constraints defining the problem, are useful when they discard some assignments that would be permitted by that minimum set of constraints. Assignments discarded by redundant constraints would also be found unfeasible by the basic constraints, but after some search that could be saved if redundant constraints are used [30]. The interaction between redundant constraints and variable ordering heuristics is relevant, to avoid that the above mentioned assignments were never found during search. Again, a deep knowledge of the problem to solve is advisable.

Redundancy can also occur on variables. *Auxiliary* variables are variables which are not strictly needed to model the problem, but their introduction allow constraints to propagate better (or to express better the problem). Although they might be considered not as important as original problem variables, they are variables in its own right and can be used to guide search.

If the chosen viewpoint does not satisfy our expectations, one can look for another viewpoint in a *reformulation* process. Classical examples of reformulation are the *dual* and the *hidden variable* transformations [2], passing from non-binary formulations into binary ones (useful in the early days). An interesting class is the so-called *permutation* problems, which have as many variables as values and solutions are permutations of such values (although not every permutation is a solution). The n-queens problem is

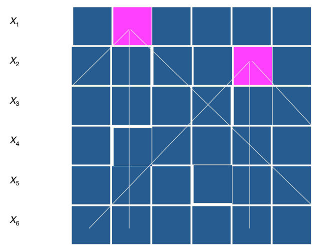

Fig. 10. A state of the 6-queens problem, from [86].

an example of this class. Permutation problems have two standard viewpoints, that exchange the roles of variables and values (in the n-queens problem, exchanging the roles of rows –variables– by columns –values–). In some cases, constraints are more easily expressed in one viewpoint than in the other. Finally, an extreme case of reformulation is to use a SAT encoding of the considered problem, to be solved by a SAT solver [93].

A powerful mechanism in modelling is to connect several (generally two) views on a problem (viewpoints), in such a way that the pruning done in one view is propagated to another view by the so called *channeling constraints*. These constraints connect the viewpoints, making available in a viewpoint what has been deduced in the other and vice versa. Viewpoints are mutually redundant, in the sense that one would suffice to achieve a correct solution, but combining them a solution can be found faster [20]. Interestingly, when two viewpoints are combined, some constraints of one (or both) viewpoints can be dropped without affecting correctness and improving performance [86]. This approach of combining viewpoints can be extended to more than two [31].

The presence of *symmetry* can also be detected and exploited in modelling. Sometimes symmetry is introduced in the modelling phase, differentiating among indistinguishable variables (i.e. in the n-queens problem, we may have a variable for the 1st queen, another for the 2nd queen, ... which is not necessary because all what we know is a set of n-queens). Reformulation and the use of set variables is a way to avoid such traps. Nevertheless, there are problems with an inherent degree of symmetry, so to avoid it some of the methods described in the Section 3.4 should be used. In particular, if we add symmetry breaking constraints before search, we can use them to generate redundant constraints [41].

In many cases, there is a function f to optimize (say minimize) under the constraints. If f returns integers, a naive approach is to solve a sequence of CSPs with a constraint requiring f lower than or equal to a parameter which decreases in steps of one ($f \leq k$, $f \leq k-1$, $f \leq k-2$ and so on). Given the first instance of the sequence without solution, the minimum of f and the solution are those of the previous instance in the sequence. Alternatively, these problems can be solved by a branch-and-bound algorithm, which each time a new solution is found with $f = k$, a new constraint is added requiring $f < k$ (so only solutions with lower values of f are considered).

Some work has been done on generating automatically several models from problem specifications, even linking them by channeling constraints [86]. The identification of constraint patterns [94], as structures that repeatedly appear in constraint networks, seems a promising modelling direction. Once these patterns are well understood, they can be successfully reused through applications.

All these ideas have been successfully used in some applications, but no idea is universally successful [40]. The number of variables, the number of constraints and the quality of their propagation seem to have a crucial impact in the amount of search needed to solve a problem. However, we should have in mind that reducing search does not always reduce run-time (because of the different overheads of the solver). A deep knowledge of the problem to solve seems to be the best advice, to be able to generate different viewpoints, redundant constraints and auxiliary variables. Some of these elements could be finally used in the problem model if they really enhance efficiency [86].

5 Constraint Programming

Programming is the process of passing from a constraint network, modelling a real-world problem, into its effective implementation in some of the existing CP environments. Obviously, you may start from scratch, programming everything from the very beginning, but nowadays this is not needed. Existing CP environments offer a number of facilities to declare variables, domains and constraints to represent your problem, to specify how search should be done and the required level of propagation (often per constraint). In addition, a library of constraints is available, many of them global offering specialized propagators already implemented. The existence of several CP environments has been crucial for the success of constraint technology. In the following, we describe some aspects of existing CP environments for finite domains (that is, we do not consider systems that work with numerical constraints and interval domains).

It is worth noting that constraint programming follows a declarative approach [38, 75]. Typically, you declare the model of your problem in terms of variables, domains and constraints. Once everything is declared, you ask for solving, but the actual solving process is not programmed by the user. Solving is done by the CP solver, which in most cases is used as a black box, that returns a solution or failure. Often, a function to be optimized can be added.

A CP environment provides facilities to define variables, domains and constraints. Variables and domains are internally represented in a rather direct way: variables are associated with domains, which are represented as sets of integers. Regarding constraints, by default they are not explicitly represented as collections of permitted tuples (to avoid memory problems and to maintain the constraint semantics), but by a collection of propagators (also known as filters or narrowing operators). A *propagator* is a function that maps domains into domains and it must be *contracting* (only remove values from the domain) and *monotonic* (if one domain is smaller than another one, this relation remains after the action of the propagator). A collection of propagators *implements* a constraint C iff they are *correct* (they do not remove assignments permitted by C) and they are *checking* C (when all variables involved in C are instantiated, the propagator must check if the tuple is permitted by C or not, reporting failure in this last case). The iterated application of propagators of the constraint set computes a reduction of domains which is the greatest mutual fixpoint of all propagators (this iterated application includes the execution of propagators for which the domains of its variables have been modified by the action of other propagators).

With this structure, the underlying search procedure (since propagators implement incomplete inference, some search is needed to achieve a solution) can be seen as a two-step procedure: (i) the application of a set of propagators until finding a fixpoint, and (ii) if no empty domain is detected, a variable is chosen on which a number of excluding alternatives are considered; each alternative is posted as a new constraint in turn, and their corresponding set of propagators is executed, applying again point (i). Step (i) is known as propagation, while step (ii) is branching (or search). The state where branching is performed is known as *choice point*. Instead of developing a search tree from variable assignments, this approach develops the search tree by adding constraints on variables (of which assigning a value is a particular case). In this sense, this approach generalizes the classical search tree of Figure 3. After reaching a fixpoint, it is of crucial importance to repeat the execution of a propagator only when the domains of the corresponding variables have changed. This is monitored by *events*, which signal changes in the domains and trigger the corresponding propagator. A number of improvements and simplifications have been suggested inside this strategy that combines search *and* propagation. They are clearly described in [81].

In the execution of the propagation step, propagators are in a list to be executed. Usually, this list is ordered putting low complexity propagators first (although the importance of the events triggering propagators is also considered). Anyway, propagator starvation (a propagator in the list waiting an unbounded amount of time to be executed) should be avoided. Although propagators have been defined as functions, in practice they have some internal, private state, that has to be stored between executions. This state is used to achieve extra functionalities (for instance, incrementality) that improve performance.

Among propagators, we mention indexicals (especially for historical reasons). An *indexical* is a propagator for a simple constraint of the form *variable is in range*. Finite domain arithmetic constraints are compiled in terms of a set of indexicals. For instance $x \leq y$ is compiled into the indexicals $x \;\; in \;\; -\infty..min(y)$ and $y \;\; in \;\; min(x)..\infty$.

The common problem of domain representation has two popular formats: range sequences and bit vectors. A range sequence is a sequence of intervals such that the domain is exactly the union of such intervals. A bit vector is formed by a set of consecutive words in memory, such that the bit in position i is set to 1 iff the ith value is in the domain. Large domains are usually represented by range sequences.

A function that CP environments repeat many times is restoration, specially applied to domains. Different techniques has been tried, such as copy, trailing and recomputation. While copy is the simplest strategy, it has the problem of memory usage. Trailing records the changes done, so they can be undone if needed. Recomputation considers repeating the execution to achieve the desired state. Obviously, this last option requires almost no extra memory, but it may last longer than the others. Nowadays, trailing is the prevailing strategy in most CP environments.

Current CP environments are divided in two main classes:

- Autonomous systems: are environments for specific programming languages which include the elements for finite domain constraint solving. Examples of such autonomous systems are Sicstus Prolog [19] and COMET [32].
- Library systems: are libraries which provide users of existing programming languages, such as C++ or Java, with the facilities to declare and work with variables, domains and constraints. Examples of library systems are Choco [58], Gecode [82] and IBM ILOG CPLEX CP Optimization Studio [53].

We also mention MiniZinc [64], an autonomous high level CP environment. MiniZinc programs are first compiled into another CP language called FlatZinc. From this language, they can be interfaced with different CP backend solvers. Here we only mention some CP environments, but there are many more. For a comprehensive list, see [81].

6 Summary

Trying to summarize all the work done in Constraint Reasoning in the last 40 years in a few pages is simply impossible. We have presented in an informal –but hopefully precise– way the most relevant achievements in this field, according to the view and formation of the author.

Most of the work described in this paper follows an algorithmic approach for solving constraint networks. We have presented search as the basic solving strategy. In systematic search, the simple BT algorithm enhanced with some intelligent backtracking techniques, appears as the kernel element for constraint solving. Also the basic blocks of local search have been described. Inference appears as an alternative way to solve constraint networks. Incomplete inference (also known as constraint propagation), specifically GAC or weaker forms of local consistency, have shown idoneous to be hybridized with systematic search strategies to achieve what is today the standard way of solving constraint networks which could be summarized in the expression $search + propagation$.

The inclusion of global constraints have been a huge step forward in achieving solutions to practical problems in reasonable time. Also symmetry exploitation has been crucial to avoid useless work in many problems showing some kind of symmetry. Relaxing some of the assumptions of the initial constraint network model, it has been extended in several dimensions generating extensions such as continuous constraints, soft constraints and distributed constraints.

Modelling, as the task of translating a real-world problem into a constraint network that could be solved in a reasonable time, appears as a difficult, hand-made task that resists automatization. Currently, a number of good practices and rules of thumb are known but a systematic way to attack modelling remains to be done. In the last part of the paper, we have described some of the elements of CP environments. There is a wide variety of them, indicating that the engineering dimension of Constraint Reasoning is in good health.

However, there are a number of other topics in Constraint Reasoning that we have not talked about, which deserve some mention. Restarts in search procedures have received some attention, and different branching strategies to develop search trees have been studied. Domain consistencies, a class of local consistencies removing values from domains (wider than arc consistency) have been shown useful in a number of cases. Complexity has been a recurrent topic, considering both the problem topology and the internal form of constraints, looking for tractability islands. In many cases, there are close connections with Operation Research (especially when optimization is involved). A substantial amount of work has been devoted to constraint languages, from the early days of logic programming to current CP environments. Another classical topic has been the study of dynamic CSPs, when one has to solve a sequence of CSP instances, each differing little from the previous one. Further extensions to the classical problem such as unbounded domains (open CSPs) and variables with structure (for instance set variables) have been analyzed. Lately, the introduction of universal quantifiers in the definition of a solution of a constraint network has been studied.

Finally, we have not talked about applications. Constraints fit quite well in the field of temporal reasoning. Planning and scheduling has been, since the very beginning, a natural field for constraint applications. In addition, there are relevant constraint applications in vehicle routing, networking and bioinformatics.

Acknowledgements

The author sincerely thanks Christian Bessiere, Christian Bliek, Javier Larrosa, Carme Torras and Mateu Villaret who read previous versions of this document providing many useful comments which largely improved the paper.

References

1. K. R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. *Proc. 15th AAAI*, pages 311–318, 1998.
3. F. Bacchus and Paul van Run. Dynamic variable ordering in csps. *Proc. 1st CP*, pages 258–275, 1995.
4. A. B. Baker. The hazards of fancy backtracking. *Proc. 12th AAAI*, pages 288–293, 1994.
5. A. Beacham, X. Chen, J. Sillito, and P. van Beek. Constraint programming lessons learned from crossword puzzles. *Proc 14th Canadian Conf. on AI*, pages 78–87, 2001.
6. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. *Global Constraint Catalog*. Working version of SICS technical report T2010:07, 2011.
7. F. Benhamou, F. Goualart, L. Granvilliers, and J.-F. Puget. Revising hull and box consistency. *Proc. 16th ICLP*, pages 124–138, 1999.
8. F. Benhamou and L. Granvilliers. *Handbook of Constraint Programming*, chapter Continuous and interval Constraints. Elsevier, 2006.
9. C. Bessiere. *Handbook of Constraint Programming*, chapter Constraint Propagation. Elsevier, 2006.
10. C. Bessiere and P. Van Hentenryck. To be or not to be ... a global constraint. *Proc. 9th CP*, pages 789–794, 2003.
11. C. Bessiere, G. Katsirelos, N. Narodyska, C. G. Quimper, and T. Walsh. Decompositions of alldifferent, global cardinality and related constrains. *Proc. 21th IJCAI*, pages 419–424, 2009.
12. C. Bessiere, P. Meseguer, E. Freuder, and J. Larrosa. On forward checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141:205–224, 2002.
13. C. Bessiere and J. C. Regin. Refining the basic arc consistency algorithm. *Proc. 17th IJCAI*, pages 309–315, 2001.
14. S. Bistareli, U. Montanari, and F. Rossi. Constraint solving over semirings. *Proc 14th IJCAI*, pages 624–630, 1995.
15. J.R. Bitner and E. M. Reingold. Backtrack programming techniques. *Communications ACM*, 18:651–656, 1975.
16. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35:268–308, 2003.
17. M. Bruynooghe. Solving combinatorial optimization problems by intelligent backtracking. *Information Processing Letters*, 11:36–39, 1981.
18. T. Carchae and J. Beck. Principles for the design of large neighborhood search. *Journal of Mathematical Modelling and Algorithms*, 8, 2009.
19. M. Carlsson et al. *SICStus Prolog User's Manual (release 4.2.0)*. Swedish Institute of Computer Science, 2011.

20. B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modelling: an experience report. *Constraints*, 4:167–192, 1999.
21. D. A. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *Proc. 11th CP*, pages 17–31, 2005.
22. M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1990.
23. J. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. *Proc. 5th KR*, pages 148–159, 1996.
24. A. Davenport, E. Tsang, C. Wang, and K. Zhu. Genet: A connectionist architecture for solving constraint statisfaton problems by iterative improvement. *Proc. 12th AAAI*, pages 325–330, 1994.
25. R. Dechter. Enhancement schems for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
26. R. Dechter. Mini-bucket: A general scheme of generating approximations in automated reasoning. *Proc. 15th IJCAI*, pages 1297–1302, 1997.
27. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
28. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
29. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
30. M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. *Proc. 8th ECAI*, pages 290–295, 1988.
31. I. Dotu, A. del Val, and M. Cebrian. Redundant modeling for the quasigroup completion problem. *Proc. 9th CP*, pages 288–302, 2003.
32. Dynadec. *Comet Tutorial (version 2.1)*. Dynamic Decision Technologies Inc, 2010.
33. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. *Proc. 7th CP*, pages 93–107, 2001.
34. B. Faltings. Arc consistency for continuous variables. *Artificial Intelligence*, 65:363–376, 1994.
35. R. E. Fikes. Ref-arf: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.
36. E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21:958–965, 1978.
37. E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29:24–32, 1982.
38. E. C. Freuder. In pursuit of the holy grail. *Constraints*, 2:57–61, 1997.
39. E. C. Freuder. Modeling: The final frontier. *Proc. PACLP*, pages 15–21, 1999.
40. A. Frisch, C. Jefferson, B. Martinez, and I. Miguel. The rules of constraint modelling. *Proc. 19th IJCAI*, pages 311–318, 2005.
41. A. Frisch, C. Jefferson, and I. Miguel. Symmetry-breaking as a prelude for implied constraints: A constraint modelling pattern. *Proc. 16th ECAI*, pages 171–175, 2004.
42. J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. *Proc. 5th IJCAI*, page 457, 1977.
43. I. Gent, K. E. Petrie, and J. F. Puget. *Handbook of Constraint Programming*, chapter Symmetry in Constraint Programming. Elsevier, 2006.
44. I. Gent and B. Smith. Symmetry breaking in constraint programming. *Proc. 14th ECAI*, pages 599–603, 2000.
45. M. L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1:25–46, 1993.
46. F. Glover and M. Laguna. *Tabu Search*. Kluver, 1997.
47. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

48. S. Golomb and L. Baumert. Backtrack programming. *Journal ACM*, 12:516–524, 1965.
49. R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
50. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.
51. P. Van Hentenryck and L. Michell. *Constraint-Based Local Search*. MIT Press, 2005.
52. H. Hoos and T. Stutzle. *Stochastic Local Search: Foundations and Aplications*. Elsevier / Morgan Kauffman, 2004.
53. IBM. *IBM ILOG CPLEX Optimization Studio documentation*. IBM, 2010.
54. K. Kask and R. Dechter. A general scheme for automatic search heuristics from specification dependencies. *Artificial Intelligence*, 129:91–131, 2001.
55. S. Kirpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
56. Z. Kiziltan. *Symmetry breaking ordering constraints*. PhD thesis, Uppsala University, Dep. Information Science, 2004.
57. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.
58. F. Laburthe and N. Jussien. *Choco Solver Documentation*. Ecole de Mines de Nantes, 2011.
59. J. Larrosa and R. Dechter. Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. *Constraints*, 8:303–326, 2003.
60. J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004.
61. Y. C. Law and J. H. M. Lee. Model induction: A new source of csp model redundancy. *Proc. 15th AAAI*, pages 54–60, 2002.
62. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
63. K. Marriot and P. J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.
64. K. Marriot, P. J. Stuckey, L. De Koninck, and H. Samulowitz. *An introduction to MiniZinc (version 1.4)*. University of Melbourne and NICTA, 2011.
65. D. McAllister, B. Selman, and H. Kautz. Evidence for invariants in local search. *Proc. 14th AAAI*, pages 321–326, 1997.
66. A. Meisels and R. Zivan. Asynchronous forward-checking for discsps. *Constraints*, 12:131–150, 2007.
67. P. Meseguer, F. Rossi, and T. Schiex. *Handbook of Constraint Programming*, chapter Soft Constraints. Elsevier, 2006.
68. S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizig conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
69. P. J. Modi, W.M. Shen, M. Tambe, and M. Yokoo. Adopt: asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2005.
70. R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
71. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
72. P. Morris. The breakout method for escaping from local minima. *Proc. 11th AAAI*, pages 40–45, 1993.
73. A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. *Proc. 19th IJCAI*, pages 266–271, 2005.
74. P. Prosser. Domain filtering can degrade intelligent backtracking search. *Proc. 13th IJCAI*, pages 262–267, 1993.
75. J.-F. Puget. Constraint programming: A great ai success. *Proc. 13th ECAI*, pages 698–675, 1998.

76. J.-F. Puget. Constraint programming next challenge: Simplicity of use. *Proc. 10th CP*, pages 5–8, 2004.
77. J. C. Regin. A filtering algorithm for constraints of difference in csps. *Proc 12th AAAI*, pages 362–367, 1994.
78. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.
79. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. *Proc. 11th ECAI*, pages 125–129, 1994.
80. T. Schiex, H. Fargier, and G. Verfaillie. Values constraint satisfaction problems: Hard and easy problems. *Proc. 14th IJCAI*, pages 631–637, 1995.
81. C. Schulte and M. Carlsson. *Handbook of Constraint Programming*, chapter Finite Constraint Programming Systems. Elsevier, 2006.
82. C. Schulte, G. Tack, and M. Lagerkvist. *Modelling and Programming with Gecode (version 3.7.1)*, 2011.
83. B. Selman, M. Johnston, A. Philips, and P. Laird. A new method for solving hard satisfiability problems. *Proc. 10th AAAI*, pages 440–446, 1992.
84. B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. *Proc. 12th AAAI*, pages 337–343, 1994.
85. M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous consistency and maintenance in distributed constraint satisfaction. *Artificial Intelligence*, 161:25–53, 2005.
86. B. Smith. *Handbook of Constraint Programming*, chapter Modelling. Elsevier, 2006.
87. B. Smith and S. Grant. Trying harder to fail first. *Proc. 13th ECAI*, pages 249–253, 1998.
88. R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.
89. E. P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
90. W. J. van Hoewe and I. Katriel. *Handbook of Constraint Programming*, chapter Global Constraints. Elsevier, 2006.
91. W. J. van Hoewe, G. Pesant, and L.-M. Rousseau. On global warming: Flow-based soft global constraints. *J. of Heuristics*, 12:347–373, 2006.
92. C. Voudouris and E. Tsang. *Handbooh of Metaheuristics*, chapter Guided Local Search. Kluwer, 2003.
93. T. Walsh. Sat $\vee$ csp. *Proc. 6th CP*, pages 441–456, 2000.
94. T. Walsh. Constraint patterns. *Proc. 9th CP*, pages 53–64, 2003.
95. D. Waltz. *The Psycology of Computer Vision*, chapter Understanding line drawings of scenes with shadows, pages 19–91. McGraw-Hill, 1975.
96. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: Asynchronous branch-and-bound dcop algorithm. *Journal of AI Research*, 38:85–133, 2010.
97. M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. Know. and Data Engin.*, pages 673–685, 1998.
98. R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11:179–197, 2006.