Constraint Programming

Course Notes 1

Nadjib Lazaar
University of Paris-Saclay
lazaar@lisn.fr

---

1 Introduction

Constraint Programming (CP) is a powerful paradigm of declarative programming that lies at the intersection of Artificial Intelligence (AI), Operations Research (OR), and Logic Programming. It is used to solve complex combinatorial problems in various fields, such as planning, resource optimization, robotics, and intelligent transportation systems. The user specifies a problem in terms of constraints, and the machine solves it, making this paradigm particularly attractive for addressing complex challenges [Rossi et al., 2006].

In CP, the designer focuses on defining the variables, domains, and constraints, while the constraint solver explores the solution space to find those that satisfy all requirements. This process enables solving problems where the relationships between variables are complex and numerous, in a more flexible and declarative way than traditional approaches.

Recent advances in machine learning (ML) and data mining have accelerated the resolution of combinatorial problems while facilitating the modeling of constraints. ML techniques can refine the search for solutions or even generate heuristics for resolution, while data mining allows extracting patterns or underlying relationships in data that can improve constraint models. This synergy between deductive methods, such as CP, and inductive methods, such as ML, opens new perspectives for solving increasingly complex problems and improving the efficiency of obtained solutions.

2 CP and NP-Hard Problems: An Efficient Solution

CP is particularly well-suited for solving combinatorial problems, especially those classified as **NP-Hard**. But what is an NP-Hard problem?

Definition of an NP-Hard Problem

A problem is called **NP-Hard** (Non-deterministic Polynomial-time Hard) when its resolution is extremely challenging, even for powerful computers. The complexity of the problem increases exponentially with the size of the problem instance. This means that for large problem sizes, it is practically impossible to find an optimal solution in a reasonable time.

In other words, as the number of variables or constraints increases, it becomes more difficult to find a solution within a reasonable timeframe. There is no known algorithm capable of efficiently solving all NP-Hard problems in polynomial time [Garey and Johnson, 1979].

NP-Hard problems are common in fields such as optimization, logistics, and planning. CP is an effective method for solving some of these problems by finding satisfactory solutions in a reasonable timeframe.

Examples of NP-Hard Problems

- **The Traveling Salesman Problem (TSP)**: Find the shortest route for a salesman who must visit a set of cities and return to the starting point. The difficulty of finding the optimal solution increases with the number of cities.
- **Graph Coloring**: Assign colors to each vertex of a graph so that no two adjacent vertices have the same color, while minimizing the total number of colors used. This problem quickly becomes complex as the number of vertices increases.
- **Task Scheduling with Constraints**: Organize a set of tasks (e.g., jobs in a factory or appointments in a calendar) while respecting various constraints (task durations, limited resources, etc.). As the number of tasks and constraints grows, the problem becomes hard to solve.

These NP-Hard problems, although complex, can be efficiently addressed using Constraint Programming, which helps find satisfactory solutions in reasonable timeframes, even as the problems become increasingly difficult to solve exactly.

3 Why CP?

Combinatorial problems can also be addressed using other techniques, such as **Integer Linear Programming** (ILP) [Schrijver, 1999] or **Propositional Satisfiability** (SAT) [Biere et al., 2021]. These methods have their own advantages and disadvantages, but CP stands out due to several characteristics that make it particularly useful in many contexts:

- **Compactness**: CP allows problems to be modeled concisely. Constraints directly express complex relationships between variables, making models simpler and more readable than traditional linear programming equations.
- **Expressiveness**: CP provides great flexibility to express a wide variety of constraints, whether linear, non-linear, global, or even logical. This allows handling highly diverse problems, from planning to designing complex systems.
- **Efficiency (often)**: Although CP involves searching in a potentially very large solution space, modern solvers are often able to exploit constraint propagation techniques and heuristic search to efficiently find solutions, even for large problems.

In comparison, ILP is often more efficient for problems where relationships between variables can be expressed linearly, but it can become cumbersome and less performant when non-linear or complex constraints are involved. Similarly, SAT is particularly suitable for discrete logical problems, but its formulation can be less intuitive than CP, especially when relationships between variables are rich and varied.

3.1 Advantages of CP

CP stands out for its advantages in several important aspects:

- **Natural Modeling**: Many problems, especially in planning, design, or optimization, can be naturally formulated in terms of constraints. This declarative model simplifies problem specification compared to other approaches like ILP.
- **Flexibility and Adaptability**: CP can easily integrate new constraints or modify existing ones without disrupting the entire model. This makes it particularly well-suited for solving evolving or unpredictable problems.
- **Separation of Search and Modeling**: In CP, problem modeling (constraints) is separate from solution searching (constraint solver), allowing designers to focus on defining requirements without worrying about the details of the solving algorithm.

In summary, Constraint Programming often proves to be a highly effective approach, particularly for complex combinatorial problems where flexibility, compactness, and expressiveness are essential for clear modeling and efficient solving.

4 Fundamental Definitions

Vocabulary

A **vocabulary** is a pair (X, D), where:

- X is a finite set of variables, denoted $X = \{x_1, x_2, \ldots, x_n\}$;
- D is a finite subset of $\mathbb{Z}$, called the *domain*, containing the possible values the variables can take.

Example: Vocabulary

Consider the problem of coloring the 13 regions of France, where each region must be colored so that two adjacent regions have different colors. We have four colors available: blue, red, green, and yellow.
To model this problem, we define the vocabulary (X, D) as follows:

- **Variables:** $X = \{x_1, x_2, \ldots, x_{13}\}$, where each x_i represents a region.
- **Domain:** $D = \{1, 2, 3, 4\}$, where: $1 =$ Red, $2 =$ Green, $3 =$ Blue, $4 =$ Yellow.

Thus, the vocabulary is defined by the pair:

$$(X, D) = (\{x_1, x_2, \ldots, x_{13}\}, \{1, 2, 3, 4\}).$$

Constraint Network

A **constraint network** $N = (X, D, C)$ is defined on a given vocabulary (X, D), where:

- X is a finite set of variables;
- D is a common or individual domain of the variables in X;
- C is a finite set of constraints.

Each constraint $c \in C$ is a pair $\langle \text{var}(c), \text{rel}(c) \rangle$, where:

- $\text{var}(c)$ is a sequence of variables from X, called the *scope of the constraint*;
- $\text{rel}(c)$ is a relation on $D^{|\text{var}(c)|}$, called the *constraint relation*, which defines the allowed combinations of simultaneous values for the variables in $\text{var}(c)$.

The **arity** of a constraint c corresponds to the size of its scope, i.e., $|\text{var}(c)|$.

For instance, for constraints defining the relationships between two adjacent regions, the arity is $|\text{var}(c)| = 2$.

Example: Constraint in Extension and Intension

In the problem of coloring the 13 regions of France, each constraint ensures that two adjacent regions cannot have the same color. Let us consider the example of adjacent regions x_1 and x_2.
The constraint c is defined as follows:

- **Scope:** $\text{var}(c) = \{x_1, x_2\}$ (concerned regions).
- **Relation:**
  - In extension: $\text{rel}(c) = \{(1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)\}$, listing all allowed pairs of colors.
  - In intension: $x_1 \neq x_2$, a more compact declarative constraint.

A constraint network C would contain a similar constraint for each pair of adjacent regions.

The representation of constraints in extension is useful for specific constraints but is memory-intensive if the domain is large. The intension representation is compact and suitable for general constraints like $x_1 \neq x_2$.

Constraint Language

A **constraint language** is a set $\Gamma = \{r_1, r_2, \ldots, r_t\}$ of relations over a subset of $\mathbb{Z}^n$.

CSP

A **Constraint Satisfaction Problem (CSP)** is the task of finding a *solution* that satisfies all constraints in a constraint network.

Example: CSP for Coloring the Regions of France

The CSP for the problem of coloring the 13 regions of France is defined as follows:

- **Variables:** $X = \{x_1, x_2, \ldots, x_{13}\}$;
- **Domain:** $D = \{1, 2, 3, 4\}$;
- **Constraints:** C contains a constraint $x_i \neq x_j$ for each pair of adjacent regions.

The objective is to find an assignment that satisfies all constraints, which is equivalent to solving a CSP.

5 Constraint Optimization Problems (COP)

A **Constraint Optimization Problem (COP)** extends a CSP by seeking not only a solution that satisfies all constraints but also one that optimizes a given objective function.

Definition: COP

A **Constraint Optimization Problem (COP)** is defined by:

- A **constraint network** (X, D, C);
- An **objective function** $f : D^n \to \mathbb{R}$, to be minimized or maximized.

The objective of a COP is to find a solution to the constraint network (X, D, C) that optimizes (minimizes or maximizes) the value of the objective function f.

Example: Minimizing the Number of Colors for the Regions of France

Revisiting the problem of coloring the 13 regions of France: in the classic version (CSP), the goal is to assign a color to each region such that two adjacent regions have different colors. In the optimized version (COP), the objective is to minimize the **total number of colors** used while respecting adjacency constraints.
The COP is defined as follows:

- **Variables:** $X = \{x_1, x_2, \ldots, x_{13}\}$, where each variable x_i represents a region.
- **Domain:** $D = \{1, 2, \ldots, k\}$, where k is the initial maximum number of possible colors.
- **Constraints:** C ensures that two adjacent regions do not share the same color: $x_i \neq x_j$ if regions i and j are adjacent.
- **Objective Function:** Minimize the total number of colors k used to color the regions.

Question

Propose the exact formulation of the **objective function** for this COP.

6 Solving a CSP

Solving a CSP involves two main steps: **search** (systematic exploration of possible solutions) and **constraint propagation** (reducing domains to facilitate the search).

6.1 Notations and Definitions

A **partial instantiation** corresponds to the assignment of values from D to a subset of variables $Y \subseteq X$, while a **complete instantiation** assigns values to all variables in X. An instantiation is said to be **inconsistent** if it violates at least one constraint in the network C. On the other hand, a partial instantiation is **locally consistent** if it satisfies all constraints involving the already instantiated variables. A **solution** to the problem is a complete instantiation that satisfies all constraints in the network.

The **search space** of a CSP is the set of all possible partial and complete instantiations of variables in X within the domain D. This space has an exponential size, $|D|^{|X|}$, making solving potentially expensive. To explore this space, a **search tree** is used, where each node represents a partial instantiation, each branch corresponds to assigning a value to a variable, and leaves represent complete instantiations, which may be solutions or failures. Solving involves traversing this tree to identify solutions or prove their absence.

6.2 Search

Search involves exploring the space of possible instantiations to find a solution.

Backtracking (BT) [Cormen et al., 2009]

Backtracking (BT) is a systematic search method that constructs a partial instantiation by assigning variables one by one. When an instantiation violates a constraint, the method backtracks to try another value.

- **Advantage:** Simple to implement.
- **Disadvantage:** Inefficient if domains are large or constraints are numerous.

Algorithm 1: Backtracking (BT)

```
Input: $\langle X, D, C\rangle$ (constraint network), $I$ (partial instantiation)
Output: true if a solution is found, otherwise false
if $I$ is a complete instantiation then
    return true ;                                  // A solution has been found
end
Select a variable $X_i \notin I$;
foreach $v \in D(X_i)$ do
    if $I \cup \{X_i \leftarrow v\}$ is locally consistent then
        if $BT(\langle X, D, C\rangle, I \cup \{X_i \leftarrow v\})$ then
            return true
        end
    end
end
return false ;                                     // No solution was found
```

6.3 Constraint Propagation

Constraint propagation aims to reduce the domains of variables by eliminating values that cannot belong to a solution. This technique, which has evolved since the 1980s, is based on the notion of local consistency. Forms such as **arc-consistency (AC)** reduce the search space by removing incompatible values. Stronger consistencies, such as path-consistency or k-consistency, achieve even greater reductions, while weaker forms are used to optimize performance. Constraint propagation often relies on iterative application of reduction rules, such as the AC3 algorithm, and is tailored to handle specific constraints as needed.

These techniques have been extensively studied and refined, contributing to a better understanding and more efficient resolution of CSPs.

AC3 Algorithm (Arc-Consistency 3) [Mackworth, 1977]

The **AC3 algorithm** ensures that each arc (x_i, x_j) in the constraint graph is **consistent**, meaning that for every value of x_i, there exists a compatible value of x_j.

- **Input:** A constraint network (X, D, C).
- **Output:** A network made arc-consistent (or detection of inconsistency if a variable's domain becomes empty).
- **Usage:** Preprocessing before search to reduce domains.

Algorithm 2: AC3 Algorithm

```
Input: $\langle X, D, C \rangle$ (CSP problem)
Output: true if arc-consistency is achieved, false otherwise
$Q \leftarrow \{(x_i, c) \mid c \in C, x_i \in \text{var}(c)\}$;
while $Q \neq \emptyset$ do
    Pick $(x_i, c) \in Q$;
    if REVISE$(x_i, c)$ then
        if $D(x_i) = \emptyset$ then
            return false;
        end
        else
            $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C, x_i, x_j \in \text{var}(c')\}$;
        end
    end
end
return true;
```

Algorithm 3: REVISE Function

```
Input: $X_i$ (variable), $c$ (constraint)
Output: true if a revision occurred, false otherwise
CHANGE $\leftarrow$ false;
foreach $v \in D(x_i)$ do
    if No allowed pair $(v, v_i)$ satisfies $c$ then
        Remove $v$ from $D(x_i)$;
        CHANGE $\leftarrow$ true;
    end
end
return CHANGE;
```

6.4 Forward Checking

Forward Checking (FC) [Haralick and Elliott, 1980]

Forward Checking (FC) is a constraint propagation method used during search, especially when assigning variables. At each step, when a variable is instantiated, FC checks the constraints between this variable and the uninstantiated ones, removing incompatible values from their domains. This ensures that, at each assignment, all constraints between instantiated and uninstantiated variables remain arc-consistent.

- **Advantage:** FC is more efficient than classical backtracking (BT) as it dynamically reduces the domains of variables, restricting the search space earlier.
- **Disadvantage:** FC cannot detect more complex inconsistencies involving multiple variables, which may only appear later in the search tree.

Algorithm 4: Forward Checking (FC) Algorithm

```
Input: $\langle X, D, C \rangle$ : CSP, $I$ : partial instantiation
Output: true if a solution is found, false otherwise
if $I$ is complete then
    return true
Select a variable $x_i \notin I$;
foreach $v \in D(x_i)$ do
    Remove all inconsistent $(v', x_j)$ with $(v, x_i)$;
    if no wipe-out occurs then
        if FC($\langle X, D, C \rangle$, $I \cup \{x_i = v\}$) then
            return true
    Restore all values pruned because of $(v, x_i)$;
return false
```

6.5 Maintaining Arc-Consistency (MAC)

Maintaining Arc-Consistency (MAC) [Gaschnig, 1974]

Maintaining Arc-Consistency (MAC) combines search and propagation by maintaining arc-consistency after each assignment. Unlike FC, MAC applies AC3 at each step to ensure that all arcs remain consistent.

- **Advantage:** Significantly reduces the search space.
- **Disadvantage:** More computationally expensive than FC.

Algorithm 5: MAC (Maintaining Arc Consistency) Algorithm

```
Input: $\langle X, D, C \rangle$ : CSP, $I$ : partial instantiation
Output: true if a solution is found, false otherwise
if $AC(\langle X, D, C \rangle)$ then
    if $I$ is complete then
        return true
    Select a variable $x_i \notin I$;
    Select a value $v \in D(x_i)$;
    if $MAC(\langle X, D, C \rangle, I \cup \{x_i = v\})$ then
        return true
    if $MAC(\langle X, D, C \cup \{x_i \neq v\} \rangle, I)$ then
        return true
return false
```

6.6 When to Use BT, FC, and MAC?

The choice between the **Backtracking (BT)**, **Forward Checking (FC)**, and **Maintaining Arc Consistency (MAC)** algorithms depends on the complexity of the problem, the size of the search space, and the requirements for constraint propagation.

- **Use BT for simple CSPs with few constraints and small domains:** The backtracking algorithm is a natural choice for simple constraint satisfaction problems. When the search space is relatively small, the variable domains are limited, and the constraints are sparse, pure backtracking can suffice. It is a generic and straightforward approach that works well in cases where brute force exploration is feasible. However, for larger search spaces, BT may become inefficient as it does not benefit from early elimination of invalid solutions.

- **Use FC for medium-sized CSPs or problems with moderate constraints:** Forward Checking (FC) is a good choice for problems with larger domains and slightly more complex constraints. FC dynamically reduces domains by eliminating incompatible values as soon as a variable is assigned, significantly reducing the search space. It is particularly effective when the problem requires local consistency checks without incurring excessive computational overhead. FC suits medium-sized problems where domain reductions at each search step can speed up finding a solution without overly increasing computational costs.

- **Use MAC for complex CSPs with many arcs and large domains, where strong consistency is required:** MAC is a more powerful method but also computationally more expensive. It is preferable in situations where the search is particularly complex, such as CSPs with numerous variables and relationships (or arcs). MAC applies strong local consistency by maintaining arc-consistency throughout the search, which significantly reduces the search space before continuing exploration. This makes it ideal for complex problems where solutions are highly constrained, and the computational investment is justifiable for achieving more efficient results.

The algorithm choice thus depends on the balance between problem complexity, search space size, and available computational resources.

6.7 Heuristics in Search

Heuristics play a crucial role in improving the efficiency of search algorithms for solving CSPs. They influence variable and value selection decisions, two critical factors for algorithm performance.

6.7.1 Variable Selection Heuristics

Variable selection heuristics determine the order in which unassigned variables are processed. The **min domain (dom)** heuristic, which selects the variable with the smallest remaining domain, is very common. Other variants like **dom+deg** (considering the degree of the variable) or **dom/deg** (dividing the domain size by the degree) can enhance performance. Heuristics based on the structure of the constraint graph, prioritizing variables that cut cycles, can also simplify the search by reducing problem complexity.

6.7.2 Value Selection Heuristics

Value selection heuristics determine the order in which values are assigned to variables. A common approach is to choose the value that maximizes the product of the sizes of the domains of other unassigned variables. This can be interpreted as an estimate of the impact of a choice on the search space. Other heuristics, such as the **sum of remaining domain sizes**, are also used, although often less effective.

Key Takeaway

Heuristics improve the efficiency of search algorithms by reducing the search space. While there is no universal solution, selecting heuristics such as those based on domain size or problem structure provides good performance in many cases. Their selection often depends on the specific characteristics of the problem to be solved.

6.8 Optimization in CP

Optimization in CP extends the standard CSP framework by introducing an **objective function** f that must be optimized, alongside constraints that must be satisfied. To achieve this, an auxiliary variable c is added and constrained as $c = f(X)$, where X represents the variables of the CSP. This is called the

objective constraint. The goal is to find a solution minimizing f. This is typically done by iteratively solving a sequence of satisfaction problems:

- Find an initial solution s satisfying all constraints.
- Add a constraint $c < f(s)$ to exclude solutions not better than s.
- Repeat the process until no solution is found. The last solution is optimal.

Constraint Propagation: To improve efficiency, constraint propagation techniques are applied to prune infeasible values of c.

6.8.1 Branch-and-Bound Algorithm

The Branch-and-Bound (B&B) algorithm combines backtracking search with pruning using bounds on the objective function.

Algorithm 6: Branch-and-Bound (B&B)

```
Input: $\langle X, D, C, f \rangle$ : Optimization CSP
Output: Optimal solution $s^*$ minimizing $f$, or null if none exists.
$bestSolution \leftarrow$ null;
$bestCost \leftarrow +\infty$;
function B&B($I$ : partial assignment);
if $I$ is complete then
    $cost \leftarrow f(I)$;
    if $cost < bestCost$ then
        $bestCost \leftarrow cost$;
        $bestSolution \leftarrow I$;
    return;
Select a variable $x_i \notin I$;
foreach $v \in D(x_i)$ do
    $I' \leftarrow I \cup \{x_i = v\}$;
    if $f(I') < bestCost$ and $I'$ satisfies $C$ then
        B&B($I'$);
return $bestSolution$;
```

Advantages: B&B systematically explores the search space, pruning sub-optimal branches, and guarantees optimality if a solution exists.

Applications: Widely used in scheduling, planning, and resource allocation where both feasibility and optimality are crucial.

7 Global Constraints in CP

Global constraints are one of the most powerful tools in the realm of CP. They encapsulate recurring patterns of constraints into higher-level abstractions, making problem modeling more expressive and computationally efficient [Beldiceanu et al., 2007].

Definition of Global Constraints

A **global constraint** is a constraint defined over an arbitrary set of variables that captures a specific property or pattern, often recurring across multiple problem domains.

Alldifferent Global Constraint Example

The `alldifferent` constraint ensures that all variables in its scope take distinct values. By aggregating several low-level constraints into a single global constraint, it is possible to leverage specialized algorithms for propagation and consistency enforcement.

7.1 Strength of Global Constraints

The power of global constraints lies in their ability to:

- **Capture High-Level Properties:** A global constraint expresses a complex property in a single, declarative statement, reducing modeling effort and increasing readability.
- **Enhance Propagation:** By using advanced filtering algorithms, global constraints can achieve higher levels of domain reduction, thus decreasing the search space.
- **Encourage Reuse:** Commonly used global constraints can be applied across various problems and domains, fostering modularity.

7.2 Decomposability and Complexity

Once a property is captured by a global constraint, it is essential to analyze it in terms of decomposition and complexity:

- **Arc-Consistency (AC) Decomposability:** A global constraint is said to be *AC-decomposable* if it can be rewritten as an equivalent set of simpler constraints (e.g., binary constraints) while maintaining arc-consistency. For instance, the Berge-acyclic property of the underlying graph structure is a key criterion for checking AC-decomposability.
- **Computational Complexity:** Many global constraints are inherently *NP-hard*, meaning their filtering algorithms are computationally expensive. When feasible, practical approximations or incomplete propagators are employed to strike a balance between efficiency and pruning power.

7.3 Propagation and Filtering Algorithms

A hallmark of global constraints is the ability to define and implement different propagators to enforce consistency levels:

- **Complete Propagators:** These achieve the strongest possible domain reductions while respecting the semantics of the constraint. However, they may be computationally expensive.
- **Partial Propagators:** These perform limited domain pruning to maintain efficiency, trading completeness for runtime performance.

For example, the `alldifferent` constraint can use algorithms based on graph matching (e.g., Hopcroft-Karp) to achieve arc-consistency. Similarly, the `sum` constraint can employ flow-based algorithms for effective propagation.

7.4 Examples of Global Constraints

- `alldifferent`: Ensures all variables in its scope have distinct values. Applications include scheduling and assignment problems.
- `sum`: Enforces a linear equation among variables, useful in resource allocation.
- `global_cardinality`: Extends `alldifferent` by bounding the frequency of each value in the domain.
- `cumulative`: Used in scheduling to ensure resource usage does not exceed a given capacity.
- `regular`: Validates sequences of variables against a finite automaton, often used in string problems or routing.

7.5 Importance in Practice

The use of global constraints allows practitioners to focus on problem modeling while relying on optimized propagators to handle the computational complexity. Their role in CP is indispensable, as they:

- Provide modular, reusable building blocks for constraint models.
- Leverage domain-specific knowledge through tailored propagation techniques.
- Reduce the gap between problem formulation and efficient resolution.

7.6 Research Directions

Active research in global constraints continues to explore:

- **New Propagators:** Developing efficient algorithms for emerging global constraints.
- **Hybrid Techniques:** Combining propagation with other solving paradigms, such as linear programming or SAT.

- **Scalability:** Ensuring propagation algorithms remain effective on large-scale instances.

Global constraints epitomize the declarative nature of CP, enabling powerful abstraction while offering practical efficiency. Their continued development remains a cornerstone of advancing CP theory and practice.

References

[Beldiceanu et al., 2007] Beldiceanu, N., Carlsson, M., Demassey, S., and Petit, T. (2007). Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62.

[Biere et al., 2021] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2021). *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, 3rd edition.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

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

[Haralick and Elliott, 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313.

[Mackworth, 1977] Mackworth, A. (1977). Consistency in networks of relations. *AI Journal*, 8:99–118.

[Rossi et al., 2006] Rossi, F., van Beek, P., and Walsh, T. (2006). *Handbook of Constraint Programming*. Volume 2 of [Rossi et al., 2006].

[Schrijver, 1999] Schrijver, A. (1999). *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley.