Handbook of Decision Making with Constraint Programming

January 29, 2024

Luca Polese
luca.polese@studio.unibo.it
Alma Mater Studiorum - Università di Bologna

Contents

1 Introduction

A combinatorial decision-making problem is a problem that is computationally difficult to solve (NP-Hard) that can only be solved by intelligent search. It's experimental, a time saver and it can reduce the cost of the environmental impact. The main purpose of CDM is that we want to make a decision within many combinations of possibilities. Each one is subject to restriction (that we will call constraints).

- we choose any solution that meet all constraints → constraint satisfaction
- we choose the best solution according to an objective → combinatorial optimization

1.1 Constraint Programming

It's a **declarative programming paradigm** for stating and solving combinatorial optimization problems.

User **models** a decision problem by formalizing:

- *unknowns* of the decision (decision variables : X_i);
- possible *values* for unknowns (domains : $D(X_i) = \{v_j\}$);
- *relations* between the unknowns (constraints : $r(X_i, X_i')$).

1.1.1 Pros of Constraint Programming

- It provides a rich language for expressing constraint and defining search procedures
- Easy to model
- Easy control of the search
- Main focus on constraints and feasibility
- Constraints reductions in the search space:
  - More constraints = more domain reductions → easier to solve.

1.1.2 Cons of Constraint Programming

- There is no focus on objective function
- There is no focus on optimality

1.1.3 Tradeoff between CP and ILP

The best choice to focus on optimality is to adopt hybrid models.

1.1.4 Solver

A **constraint solver** finds a *solution* to the *model* (or proves that no solution exists) by assigning a value to every variable ($X_i \leftarrow v_j$) via a search algorithm.

The constraint solver enumerates all value combinations (variable-value) via backtracking tree search and examines constraints.

1.2 Constraint Solver

- *Enumerates* all possible variable-value combinations via a systematic backtracking tree search.
- *Guesses* a value for each variable.
- During search, *examines* the constraints to *remove incompatible values* from the domains of the future (unexplored) variables, via **propagation**
- *Shrinks* the domains of the **future variables**

2 Modeling

2.1 Formalization as a Constraint Satisfaction Problem

A CSP P is a triple $P =< X, D, C >$ where

- X is a set of decision variables $\{X_1, ..., X_n\}$,
- D is a set of domains $\{D_1, ..., D_n\}$ for X so that D_i is a set for X_i (non binary, with finite domain),
- C is a set of constraints $\{C_1, ..., C_n\}$ so that C_i is a relation over $X_j, ..., X_k$ denoted as $C(X_j, ..., X_k)$.

A solution to a CSP P is an n-tuple $A =< a_1, a_2, \ ... \ , a_n >$ where every a_i is an assignment of value to the variable X_i which satisfies all constraints simultaneously.
If the set of solutions is empty, the CSP is *unsatisfactory.*

2.2 Constraint Optimization Problems

CSP enhanced with an optimization criterion
Formally, $< X, D, C, f >$ where f is the formalization of the optimization criterion as an objective variable.
Our goal is to minimize/maximize f.

2.3 Constraints

We can use two different approaches to determine the constraints:

- Any constraint can be expressed by listing all allowed combinations.
  - General but inconvenient and inefficient with large domains.
- Usually it's better to use relations between objects.
  - Less general, but more compact and clear.

2.3.1 Properties of constraints

- The order of imposition does not matter
- Non-directional: a constraint between X and Y can be used on Y and X and vice versa
- Rarely independent, usually variables are shared between different constraints.

2.3.2 Types of constraint

Type	**Example**
Algebraic expressions	$x_1 > x_2$
Extensional constraints (*table* constraints)	$(X, Y, Z) \in \{(a, a, a), (b, b, b)\}$
Variables as subscripts (*element* constraints)	$Y = \text{cost}[X]$ (where cost is an array of parameters)
Logical relations	$(X < Y) \vee (Y < Z) \to C$
Global constraints	alldifferent$([X_1, X_2, X_3])$
Meta-constraints	$\sum_i (X_i > t_i) \leq 5$ (at most 5 variables should satisfy this constraint)

2.4 Modeling is critical

- Choice of variables and domains defines the search space size:
  $|D(X_1)| \times |D(X_2)| \times ... \times |D(X_n)| \to$ exponential!
- Choice of constraints defines:

- how search space can be reduced
- how search can be guided.

2.5 Symmetry in CSPs

> *Search can revisit equivalent states over and over again.*
>
> — Handbook of Constraint Programming

By **symmetry** we mean that:

> *Given a solution, which by definition satisfies all the constraints, we can find a* ***new solution*** *by applying any* ***symmetry*** *to the first solution we find. [dots] Why is symmetry important? The main reason is that we can* ***exploit symmetry*** *to* ***reduce*** *the amount of* ***search*** *needed to solve the problem.*
>
> — Handbook of Constraint Programming

A CSP can create many symmetrically equivalent search states each of which leads to a solution/failure and that will have many symmetrically equivalent states.

This is bad when we're proving optimality, infeasibility or looking for all solutions.

2.5.1 Permutation

Defined over a discrete set S as a 1-1 function π: $S \to S$, intuitively we re-arrange a set of elements.

2.5.2 Variable Symmetry

It's a permutation π of the variable indices s.t. for each (un)feasible (partial) assignment, we can re-arrange the variables according to π and obtain another (un)feasible (partial) assignment.

- Intuitively: permuting variable assignments.
- π identifies a specific symmetry.

2.5.3 Value Symmetry

It's a permutation π of the values s.t.for each (un)feasible (partial) assignment, we can re-arrange the values according to π and obtain another (un)feasible (partial) assignment.

- Intuitively: permuting values.
- π identifies a specific symmetry.

2.5.4 Symmetry Breaking Constraints

If we break the previous symmetries, then we reduce the set of solutions and search space.

- Not logically implied by the constraints of the problem.
- Attention: at least one solution from each set of symmetrically equivalent solutions must remain.

2.5.5 Dual Viewpoint

- Viewing a problem P from different perspectives may result in different models for P. Each model yields the same set of solutions. Each model exhibits in general a different representation of P. Can be hard to decide which is better.

2.5.6 Combined model

When we have multiple models that have relevant pros and cons we can opt for combining them. This way we can keep all the models and use channeling constraints to maintain consistency between the variables of the two models.

Benefits:

- Easier to express constraints
- Enhanced constraint propagation
- More options for search variables.

3 Constraint Propagation & Global Constraints

3.1 Local Consistency

A form of inference which detects inconsistent partial assignments. It's *local*, because we examine individual constraints.

Two popular local consistencies, domain based (they can change the domain values):

- Generalized Arc Consistency (GAC) or Domain Consistency
- Bounds Consistency (BC)

They detect inconsistent partial assignments of the form $X_i = j$, hence: j can be removed from $D(X_i)$ via propagation and propagation can be implemented easily.

3.1.1 Arc Consistency

> *This is indeed a very simple and natural concept that guarantees every value in a domain to be consistent with every constraint.*
>
> — Handbook of Constraint Programming

3.1.2 Generalized Arc Consistency (GAC)

A constraint C on k variables $C(X_1, ..., X_k)$ gives the set of allowed combinations of values.

$$C \subseteq D(X_1) \times ... \times D(X_k)$$

Each allowed tuple $(d_1, ..., d_k) \in C$ where $d_i \in X_i$ is a support for C.

$C(X_1, ..., X_k)$ is GAC iff: $\forall X_i \in \{X_1, ..., X_k\}$, $\forall v \in D(X_i)$, v belongs to a support for C. We call it just Arc Consistency (AC) when $k = 2$.

A CSP is GAC iff all its constraints are GAC.

3.1.3 Bounds Consistency (BC)

Defined for totally ordered domains.

BC relaxes the domain of X_i from $D(X_i)$ to

$$[\min(X_i) ... \max(X_i)]$$

A bound support is a tuple $(d_1, ..., d_k) \in C$ where $d_i \in [\min(X_i) ... \max(X_i)]$.

$C(X_1, ..., X_k)$ is BC iff: $\forall X_i \in \{X_1, ..., X_k\}$, $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

Disadvantages:

- BC might not detect all GAC inconsistencies in general.
- We need to search more.

Advantages:

- Might be easier to look for a support in a range than in a domain.
- Achieving BC is often cheaper than achieving GAC.

Of interest in arithmetic constraints defined on integer variables with large domains. Achieving BC is enough to achieve GAC for monotonic constraints.

3.1.4 GAC = BC

- All values of $D(X) \leq \max(Y)$ are GAC.
- All values of $D(Y) \geq \min(X)$ are GAC.
- Enough to adjust $\max(X)$ and $\min(Y)$.

$$\max(X) \leq \max(Y) \wedge \min(X) \leq \min(Y)$$

3.2 Constraint Propagation

A local consistency notion defines properties that a constraint C must satisfy **after constraint propagation**. The only requirement is to achieve the required property on C.

3.2.1 Propagation Algorithms

A propagation algorithm achieves a certain level of consistency on a constraint C by removing the inconsistent values from the domains of the variables in C. The level of consistency depends on C.

- GAC if an efficient propagation algorithm can be developed
- Otherwise BC or a lower level of consistency

When solving a CSP with multiple constraints:

- propagation algorithms interact and wake up an already propagated constraint to be propagated again
- in the end, propagation reaches a fixed-point and all constraints reach a level of consistency;

The whole process is referred as **constraint propagation**.

3.2.2 Properties of Propagation Algorithms

It may not be enough to remove inconsistent values from domains once. A propagation algorithm must wake up again when necessary, otherwise may not achieve the desired local consistency property. Multiple events can trigger a constraint propagation:

- when the domain of a variable changes (for GAC)
- when the domain bounds of a variable changes (for BC)
- when a variable is assigned a value.

3.2.3 Complexity of Propagation Algorithms

Assume $|D(X_i)| = d$. Following the definition of the local consistency property: one time AC propagation on a $C(X_1, X_2)$ takes $O(d^2)$ time.

3.3 Specialized Propagation

Propagation specific to a given constraint.

Advantages:

- Exploits the constraint semantics
- Potentially much more efficient than a general propagation approach.

Disadvantages:

- Limited use.

- Not always easy to develop one.

Worth developing for recurring constraints.

3.4 Global Constraints

Capture complex, non-binary and recurring combinatorial substructures.

Embed specialized propagation which exploits the substructure.

Modelling benefits:

- Reduce the gap between the problem statement and the model.
- May allow the expression of constraints that are otherwise not possible to state using primitive constraints (semantic).

Solving benefits:

- Strong inference in propagation (operational)
- Efficient propagation (algorithmic).

3.4.1 Counting Constraints

Restrict the number of variables satisfying a condition or the number of times values are taken.

`Alldifferent Constraint`

alldifferent$([X_1, X_2, ..., X_k])$ iff $X_i \neq X_j$ for $i < j \in \{1, ..., k\}$

- permutation constraint with $|D(X_i)| = k$

`Nvalue Constraint`

Constrains the number of **distinct values** assigned to the variables.
nvalue$([X_1, X_2, ..., X_k], N)$ iff $N = |\{X_i \; |1 \leq i \leq k\}|$

- `alldifferent` when $N = k$

`Global Cardinality Constraint`

Constrains the number of times **each value is taken** by the variables.
gcc$([X_1, X_2, ..., X_k], [v_1, ..., v_m], [O_1, ..., O_m])$ iff
$\forall j \in \{1, ..., m\} \; O_j = |\{X_i \mid X_i = v_j, 1 \leq i \leq k\}|$

- `alldifferent` when $O_j \leq 1$.

`Among Constraint`

Constrains the number of variables taken from a given set of values.
among$([X_1, X_2, ..., X_k], s, N)$ iff $N = |\{i \; |X_i \in s, 1 \leq i \leq k\}|$

- N can also be in interval $[l, ..., u]$

3.4.2 Sequencing Constraints

Ensure a sequence of variables obey certain patterns.

`Sequence/AmongSeqConstraint`

Constrains the number of values taken from a given set in any subsequence of q variables.
sequence$(l, u, q, [X_1, X_2, ..., X_k], s)$ iff
among$\left([X_i, X_{i+1}, ..., X_{i+q-1}], s, l, u\right) \forall i$ s.t. $1 \leq i \leq k - q + 1$

3.4.3 Scheduling Constraints

Help schedule tasks with respective release times, duration, and deadlines, using limited resources in a time interval.

Disjunctive Resource Constraint

Requires that tasks do not overlap in time. Also known as `noOverlap` constraint.
Given tasks $t_1, ..., t_k$ each associated with a start time S_i and duration D_i: disjunctive($[S_1, ..., S_k], [D_1, ..., D_k]$) iff $\forall i < j$ s.t. $\left(S_i + D_i \leq S_j\right) \lor \left(S_j + D_j \leq S_i\right)$

Cumulative Resource Constraint

Constrains the usage of a shared resource.
Given tasks $t_1, ..., t_k$ each associated with a start time S_i, duration D_i, resource requirement R_i, and a resource with a capacity C:
cumulative($[S_1, ..., S_k], [D_1, ..., D_k], [R_1, ..., R_k], C$) iff $\sum_{i \mid S_i \leq u < S_i + D_i} R_i \leq C \;\forall u \in D$

3.4.4 Ordering Constraints

Enforce an ordering between the variables or the values.

Lexicographic Ordering Constraint

It requires a sequence of variables to be lexicographically less than or equal to another sequence of variables.

lex $\leq$ ($[Y_1, Y_2, ..., Y_k], [Z_1, Z_2, ..., Z_k]$) holds iff:

$$\begin{aligned}
Y_1 \leq Z_1 \land \\
(Y_1 = Z_1 \rightarrow Y_2 \leq Z_2) \land \\
(Y_1 = Z_1 \land Y_2 = Z_2 \rightarrow Y_3 \leq Z_3) \land \\
... \\
(Y_1 = Z_1 \land Y_2 = Z_2....Y_k - 1 = Z_k - 1 \rightarrow Y_k \leq Z_k)
\end{aligned}$$

Value Precedence Constraint

Requires a value to precede another value in a sequence of variables.
value_precede$\left(v_{j1}, v_{j2}, [X_1, X_2, ..., X_k]\right)$ holds iff:
$\min\left\{i \mid X_i = v_{j1} \lor i = k + 1\right\} < \min\left\{i \mid X_i = v_{j2} \lor i = k + 2\right\}$.

3.5 Specialized Propagation for Global Constraints

We define two main approaches to develop specialized propagation for global constraints:

- constraint decomposition
- dedicated ad-hoc algorithm.

3.6 Constraint Decomposition

A global constraint is decomposed into smaller and simpler constraints, each of which has a known propagation algorithm.

Propagating each of the constraints gives a propagation algorithm for the original global constraint. Effective and efficient method for some global constraints.

3.6.1 A decomposition of `among`

Decomposition as a conjunction of logical constrains and a sum constraint.

- B_i with $D(B_i) = \{0, 1\}$ for $1 \leq i \leq k$
- $C_i : B_i = 1$ iff $X_i \in s$ for $1 \leq i \leq k$
- $C_{k+1} : \sum_i B_i = N$

AC(C_i) $\forall i$ and BC$\left(\sum_i B_i = N\right)$ ensures GAC on among.

3.6.2 A decomposition of `lex`

lex $\leq ([X_1, X_2, ..., X_k], [Y_1, Y_2, ..., Y_k])$

Decomposition as a conjunction of disjunctions

B_i with $D(B_i) = \{0, 1\}$ for $1 \leq i \leq k + 1$ to indicate the vectors have been ordered by position $i - 1$.

- $B_i = 0 - C_i$:

$(B_i = B_{i+1} = 0 \wedge X_i = Y_i) \qquad \vee$

$(B_i = 0 \wedge B_{i+1} = 1 \wedge X_i < Y_i) \vee$

$(B_i = B_{i+1} = 1)$ for $1 \leq i \leq k$

GAC(C_i) for all i ensures GAC on lex $\leq$.

Decomposition as a conjunction of implications

AC on the decomposition is weaker than GAC on lex $\leq$. lex $\leq$ is not GAC but the decomposition does not prune anything.

3.6.3 Constraint Decompositions

May not always provide an effective propagation.

Often GAC on the original constraint is stronger than (G)AC on the constraints in the decomposition.

3.6.4 A Decomposition of `alldifferent`

Decomposition as a conjunction of difference constraints.

$C_{ij} : X_i \neq X_j$ for $i < j \in \{1, ..., k\}$ AC$(C_{ij}) \forall i < j$ is weaker than GAC on `alldifferent`.

Alldifferent is not GAC but the decomposition does not prune anything.

3.6.5 A Decomposition of `sequence`

Decomposition as a conjunction of among constraints.

$C_i : \text{among}([X_i, X_{i+1}, ..., X_{i+q-1}], s, l, u)$ for $1 \leq i \leq k - q + 1$

GAC(C_i) for all i is weaker than GAC on sequence.

Sequence is not GAC but the decomposition does not prune anything.

3.6.6 Decomposition vs Ad-hoc Algorithm

Even if a decomposition is effective, may not always provide an efficient propagation.
Often propagating a constraint via an ad-hoc algorithm is faster than propagating the (many) constraints in the decomposition.

3.6.7 Incremental Computation

A propagation algorithm is often called multiple times, so we don't want to re-compute everything each time.

Incremental computation can improve efficiency.

- At the first call, some partial results are cached.
- At the next invoke, we exploit the cached data.

3.6.8 Dedicated Propagation Algorithms

Dedicated ad-hoc algorithms provide effective and efficient propagation. Often:

- GAC is maintained in polynomial time
- many more inconsistent values are detected compared to the decompositions
- computation is done incrementally.