

Constraint Programming

– Practical Introduction –

Christophe Lecoutre

CRIL-CNRS UMR 8188
Université d'Artois
Lens, France

Tutorial at PFIA'25 – June 2025

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion

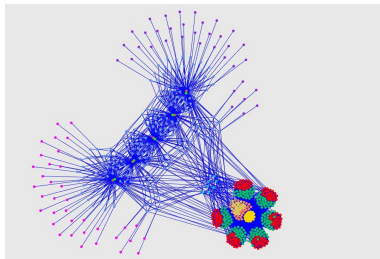
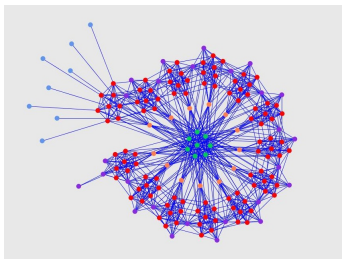
Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion

Constraint Programming

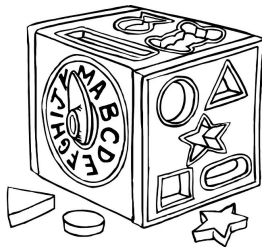
is about
modeling and **solving**

Problems under Constraints



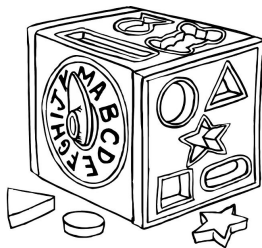
(images by John Slaney from the [G12 Visualisation Project](#))

Certainly, the first Problem you Solved!



This is a matching problem !

Certainly, the first Problem you Solved!



This is a matching problem !

Quite often, you face Optimization Problems!

A witch can elaborate two kinds of magic potions:

- the former for inspiring love
- the latter for restoring youth

Theses potions are composed of dribble of toads (U_t), wings of dragons (U_d) and powder of spiders (U_s), with the following proportions:

- 1 potion for love requires 3 U_t , 1 U_d and 1 U_s
- 1 potion for youth requires 2 U_t , 3 U_d and 2 U_s



Quite often, you face Optimization Problems!

A witch can elaborate two kinds of magic potions:

- the former for inspiring love
- the latter for restoring youth

Theses potions are composed of dribble of toads (U_t), wings of dragons (U_d) and powder of spiders (U_s), with the following proportions:

- 1 potion for love requires 3 U_t , 1 U_d and 1 U_s
- 1 potion for youth requires 2 U_t , 3 U_d and 2 U_s



Quite Often, Optimization Problems!

The witch possesses 800 U_t , 700 U_d and 400 U_s . Selling magic potions brings in:

- 4 crowns for each love potion
- 5 crowns for each youth potion

Considering her available ingredients, how many love potions (x) and youth potions (y) should be prepared by the witch in order to optimize her benefit?

$$\begin{array}{ll}\text{Maximize :} & 4x + 5y \\ \text{such that:} & \left\{ \begin{array}{l} 3x + 2y \leq 800 \\ x + 3y \leq 700 \\ x + 2y \leq 400 \end{array} \right.\end{array}$$

Quite Often, Optimization Problems!

The witch possesses 800 U_t , 700 U_d and 400 U_s . Selling magic potions brings in:

- 4 crowns for each love potion
- 5 crowns for each youth potion

Considering her available ingredients, how many love potions (x) and youth potions (y) should be prepared by the witch in order to optimize her benefit?

$$\begin{array}{ll} \text{Maximize :} & 4x + 5y \\ \text{such that:} & \left\{ \begin{array}{l} 3x + 2y \leq 800 \\ x + 3y \leq 700 \\ x + 2y \leq 400 \end{array} \right. \end{array}$$

Quite Often, Optimization Problems!

The witch possesses 800 U_t , 700 U_d and 400 U_s . Selling magic potions brings in:

- 4 crowns for each love potion
- 5 crowns for each youth potion

Considering her available ingredients, how many love potions (x) and youth potions (y) should be prepared by the witch in order to optimize her benefit?

$$\begin{array}{ll}\text{Maximize :} & 4x + 5y \\ \text{such that:} & \left\{ \begin{array}{l} 3x + 2y \leq 800 \\ x + 3y \leq 700 \\ x + 2y \leq 400 \end{array} \right.\end{array}$$

Your first PyCSP³ Model

```
from pycsp3 import *

# x is the number of magic love potions
x = Var(range(400))

# y is the number of magic youth potions
y = Var(range(400))

satisfy(
    3*x + 2*y <= 800,
    x + 3*y <= 700,
    x + 2*y <= 400
)

maximize(
    4*x + 5*y
)
```

This is an Integer Linear Program!

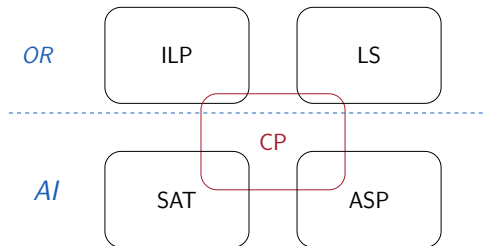


What is the point of using CP?

Solving Constrained Combinatorial Problems

With a focus on integer variables, several paradigms coming from:

- Operations Research : Integer Linear Programming and Local Search
- Artificial Intelligence : Constraint Programming, Satisfiability Testing and Answer Set (Logic) Programming



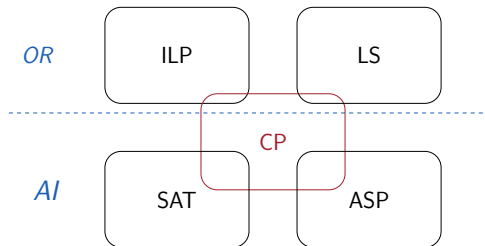
Important:

- each paradigm has its strengths and weaknesses
- one asset of CP: high level of declarativity/flexibility

Solving Constrained Combinatorial Problems

With a focus on integer variables, several paradigms coming from:

- Operations Research : Integer Linear Programming and Local Search
- Artificial Intelligence : Constraint Programming, Satisfiability Testing and Answer Set (Logic) Programming



Important:

- each paradigm has its strengths and weaknesses
- one asset of CP: high level of declarativity/flexibility

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

You use Constraint Programming every day!

or your grand-parents!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Fill the empty cells with values in 1..9

While considering that:

- each row must contain different numbers
- each column must contain different numbers
- each bloc (3×3 square) must contain different numbers

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Variables

Fill the empty cells with values in 1..9

While considering that:

- each row must contain different numbers
- each column must contain different numbers
- each bloc (3×3 square) must contain different numbers

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Variables

Domains

Fill the empty cells with values in 1..9

While considering that:

- each row must contain different numbers
- each column must contain different numbers
- each bloc (3×3 square) must contain different numbers

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

• Variables

• Domains

Fill the empty cells with values in 1..9

• Constraints

While considering that:

- each row must contain different numbers
- each column must contain different numbers
- each bloc (3×3 square) must contain different numbers

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Solution: assignment of a value to each variable such that no constraint is violated

You use Constraint Programming every day!

Sudoku

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2				7		
	9			3			8	
2			8		4			7
	1		9		7		6	

Solution: assignment of a value to each variable such that no constraint is violated

✳ A Sudoku puzzle is a very simple (instance of a) Constraint Satisfaction Problem

You use Constraint Programming every day!

Sudoku

123 456 789	2	123 456 789	5	123 456 789	1	123 456 789	9	123 456 789
8	123 456 789	123 456 789	2	123 456 789	3	123 456 789	123 456 789	6
123 456 789	3	123 456 789	123 456 789	6	123 456 789	123 456 789	7	123 456 789
123 456 789	123 456 789	1	123 456 789	123 456 789	123 456 789	6	123 456 789	123 456 789
5	4	123 456 789	123 456 789	123 456 789	123 456 789	123 456 789	1	9
123 456 789	123 456 789	2	123 456 789	123 456 789	123 456 789	7	123 456 789	123 456 789
123 456 789	9	123 456 789	123 456 789	3	123 456 789	123 456 789	8	123 456 789
2	123 456 789	123 456 789	8	123 456 789	4	123 456 789	123 456 789	7
123 456 789	1	123 456 789	9	123 456 789	7	123 456 789	6	123 456 789

Solving the puzzle with CP means:

- reasoning with constraints in order to prune values in variable domains
- assigning variables in order to construct solutions

Sudoku in Practice

Sudoku grids in: <https://www.cril.fr/~lecoutre/data/sudoku.txt>



Modeling/Solving in just 10'

Let us start!

Constraint Programming (CP) is a general framework proposing simple, general and efficient algorithmic solutions to problems under constraints.

CP is attractive because there is a clear separation between:

- on the one hand, its formalism that makes easy the representation of many problems
- on the other hand, a large pool of algorithms and heuristics to find solutions

Constraint Programming

Constraint Programming (CP) is a general framework proposing simple, general and efficient algorithmic solutions to problems under constraints.

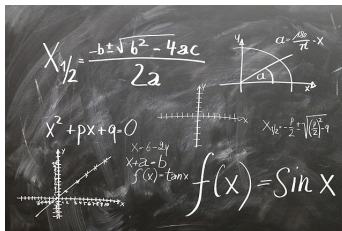
CP is attractive because there is a clear separation between:

- on the one hand, its formalism that makes easy the representation of many problems
- on the other hand, a large pool of algorithms and heuristics to find solutions

Modeling

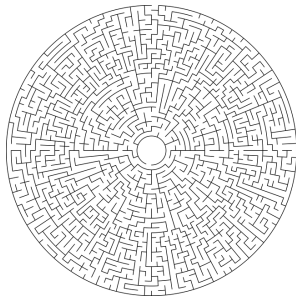
There are then two main stages with CP:

- 1 In a first stage, called modeling, the problem must be formally represented by means of variables, constraints, and possibly objective functions.



Ideally, this stage is purely declarative, but in practice a limited process of programming may be required (e.g., with a logic or object language).

-
- 2 In a second stage, called solving, the problem modelled by the user must be tackled by a software tool in order to automatically obtain one solution, all solutions, an optimal solution, ...



Constraint Satisfaction

The **Constraint Satisfaction Problem** (CSP) resides at the core of constraint programming. An *instance* of this problem is represented by a **constraint network** (CN).

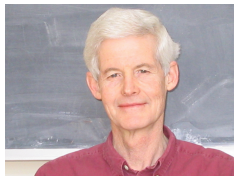


Note that SAT is closely related to CSP:

- variables are Boolean
- constraints are clauses (disjunctions of variables and their negations)

Untractability

No polynomial algorithm is known for both decision problems CSP and SAT.



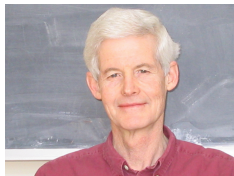
Stephen Cook showed that SAT is a NP-complete problem

Remark.

In practice, people try to find efficient algorithms for a large range of problems by exploiting their structures. This is a motivating challenge.

Untractability

No polynomial algorithm is known for both decision problems CSP and SAT.



Stephen Cook showed that SAT is a NP-complete problem

Remark.

In practice, people try to find efficient algorithms for a large range of problems by exploiting their structures. This is a motivating challenge.

Bibliography

- Foundations of constraint satisfaction (Tsang, 1993)
- Principles of Constraint Programming (Apt, 2003)
- Constraint Processing (Dechter, 2003)
- Constraint-based local search (van Hentenryck & Michel, 2005)
- Handbook of Constraint Programming (Rossi *et al.*, 2006)
- Constraint Propagation (Bessiere, 2006)
- Constraint Networks (Lecoutre, 2009)
- Global constraints: a survey (Régin, 2011)
- Integrated Methods for Optimization (Hooker, 2012)
- The art of Computer Programming (Vol. 4, Fasc. 7) (Knuth, 2025)



Outline

- 1 Introduction to CP
- 2 CP Applications**
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion

CP Success Stories: CP Has Landed on the Comet

On June 13th 2015, the robot-lab Philae woke up on the comet 67P/Churyumov-Gerasimenko to resume a series of experiments.



Some plans executed by Philae are modelled and solved using constraint programming technology. Several dedicated algorithms were developed by the Operations Research & Constraints group of the LAAS-CNRS lab.

CP Success Stories: CP Has Landed on the Comet

On June 13th 2015, the robot-lab Philae woke up on the comet 67P/Churyumov-Gerasimenko to resume a series of experiments.



Some plans executed by Philae are modelled and solved using constraint programming technology. Several dedicated algorithms were developed by the Operations Research & Constraints group of the LAAS-CNRS lab.

CP Success Stories: Pythagorean Triples Problem

Problem: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$?

A prize for the solution was offered by Ronald Graham over two decades ago.

This problem has been solved by a hybrid SAT approach, employing both look-ahead and CDCL solvers.

This is the largest computed-aided Mathematics proof (200TB) ever presented in the literature.

CP Success Stories: Pythagorean Triples Problem

Problem: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$?

A prize for the solution was offered by Ronald Graham over two decades ago.

This problem has been solved by a hybrid SAT approach, employing both look-ahead and CDCL solvers.

This is the largest computed-aided Mathematics proof (200TB) ever presented in the literature.

CP Success Stories: Pythagorean Triples Problem

Problem: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$?

A prize for the solution was offered by Ronald Graham over two decades ago.

This problem has been solved by a hybrid SAT approach, employing both look-ahead and CDCL solvers.

This is the largest computed-aided Mathematics proof (200TB) ever presented in the literature.

CP Success Stories: Pythagorean Triples Problem

Problem: Can the set $\mathbb{N} = \{1, 2, \dots\}$ of natural numbers be divided into two parts, such that no part contains a triple (a, b, c) with $a^2 + b^2 = c^2$?

A prize for the solution was offered by Ronald Graham over two decades ago.

This problem has been solved by a hybrid SAT approach, employing both look-ahead and CDCL solvers.

This is the largest computed-aided Mathematics proof (200TB) ever presented in the literature.

Constraint programming techniques applied to contemporary music composition

Two illustrative applications (with the courtesy of Charlotte Truchet, LINA).

- ① A problem on harmony (Georges Bloch). Given a fixed chord, we have the following constraints:
 - minimize EstradaDistance(chord[i], FixedChord)
 - minimize GeorgesDistance(VF(chord[i]), VF(chord[i+1]))
- ② Asynchronous rhythms (Mauro Lanza). Each pattern is played repetitively on one voice.
 - Variables: rhythmical patterns of fixed durations $d_1, \dots, d_n$.
 - Constraints: no simultaneous onsets between two voices, for a given duration D.

Constraint programming techniques applied to contemporary music composition

Two illustrative applications (with the courtesy of Charlotte Truchet, LINA).

- ① A problem on harmony (Georges Bloch). Given a fixed chord, we have the following constraints:
 - minimize EstradaDistance(chord[i], FixedChord)
 - minimize GeorgesDistance(VF(chord[i]), VF(chord[i+1]))
- ② Asynchronous rhythms (Mauro Lanza). Each pattern is played repetitively on one voice.
 - Variables: rhythmical patterns of fixed durations $d_1, \dots, d_n$.
 - Constraints: no simultaneous onsets between two voices, for a given duration D.

Constraint programming techniques applied to contemporary music composition

Two illustrative applications (with the courtesy of Charlotte Truchet, LINA).

- ① A problem on harmony (Georges Bloch). Given a fixed chord, we have the following constraints:
 - minimize EstradaDistance(chord[i], FixedChord)
 - minimize GeorgesDistance(VF(chord[i]), VF(chord[i+1]))
- ② Asynchronous rhythms (Mauro Lanza). Each pattern is played repetitively on one voice.
 - Variables: rhythmical patterns of fixed durations $d_1, \dots, d_n$.
 - Constraints: no simultaneous onsets between two voices, for a given duration D .

Constraint programming techniques applied to contemporary music composition

Two illustrative applications (with the courtesy of Charlotte Truchet, LINA).

- ① A problem on harmony (Georges Bloch). Given a fixed chord, we have the following constraints:
 - minimize `EstradaDistance(chord[i], FixedChord)`
 - minimize `GeorgesDistance(VF(chord[i]), VF(chord[i+1]))`
- ② Asynchronous rhythms (Mauro Lanza). Each pattern is played repetitively on one voice.
 - Variables: rhythmical patterns of fixed durations $d_1, \dots, d_n$.
 - Constraints: no simultaneous onsets between two voices, for a given duration D .

Classical CP Applications: Flow Shop Scheduling

Given n machines and m jobs, knowing that:

- each job requires exactly n operations to be executed
- the i th operation of a job must be executed on the i th machine
- no machine can perform more than one operation simultaneously
- for each operation of each job, execution time is specified

Find the best schedule, i.e. the one with the shortest possible total job execution makespan.

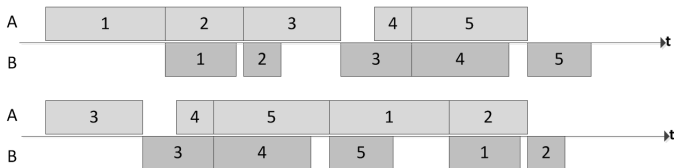


Figure: Example of (no-wait) flow-shop scheduling with 5 jobs on 2 machines A and B. A comparison of total makespan is given for two different job sequences.

Classical CP Applications: Car Sequencing

Given an assembly line for producing cars with various options, knowing that:

- different stations install different options
- stations can handle at most a certain percentage of the passing cars
- cars requiring a certain option must not be bunched together

Find a production order respecting the capacity constraints of the stations.

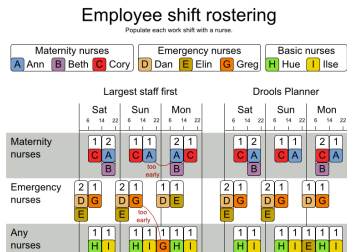


Classical CP Applications: Nurse Rostering

Given hospital shifts to be taken by nurses, knowing that:

- nurses have various qualifications
- the hospital must respect some working/resting patterns
- each hospital service has some needs

Find an optimal assignment of nurses to shifts.



Classical CP Applications: Vehicle Routing

Given orders made by some costumers, knowing that:

- a fleet of vehicles is available
- a depot stores products

Find the best route of each delivery vehicle.



Classical CP Applications: Container Ship Loading

Given containers to be loaded on some ships, knowing that:

- some cranes are available
- there are specific requests on containers to be loaded

Find the fastest loading schedule



No CP used here!



Many other application domains:

- Configuration
- Bio-informatics
- Planning
- Airport Scheduling
- Model Checking
- Data Mining
- ...

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite**
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion

Integrated Constraint Programming Suite

Suite composed of three components:

- PyCSP3, a Python-based modeling library (pycsp.org)
- XCSP3, a format preserving the structure of problem instances
- ACE, a constraint solver in Java

Interest of this approach:

- fast modeling with a dedicated Python interface
- easy reading and understanding of the format
- tight control of solving mechanisms

Intelligibility at every stage of the modelling and solving process

Modeling Languages/Libraries

Modeling languages/libraries can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

- ① identifying the **parameters**, i.e., the structure of the data
- ② writing the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages/Libraries

Modeling languages/libraries can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

- ① identifying the **parameters**, i.e., the structure of the data
- ② writing the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages/Libraries

Modeling languages/libraries can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

- ① identifying the **parameters**, i.e., the structure of the data
- ② writing the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

Modeling Languages/Libraries

Modeling languages/libraries can be used to represent problems, using some form of control and abstraction.

Typically, a model captures a family of problem instances, by referring to some parameters representing the data. Building a model for a problem involves:

- 1 identifying the **parameters**, i.e., the structure of the data
- 2 writing the **model**, by taking the parameters into account, and using an appropriate (high-level) language

Once we have a model, we still have to provide the effective **data** for each specific instance to be treated.

🔗 Let us illustrate this with the academic problem “Warehouse Location”

Warehouse Location in Practice

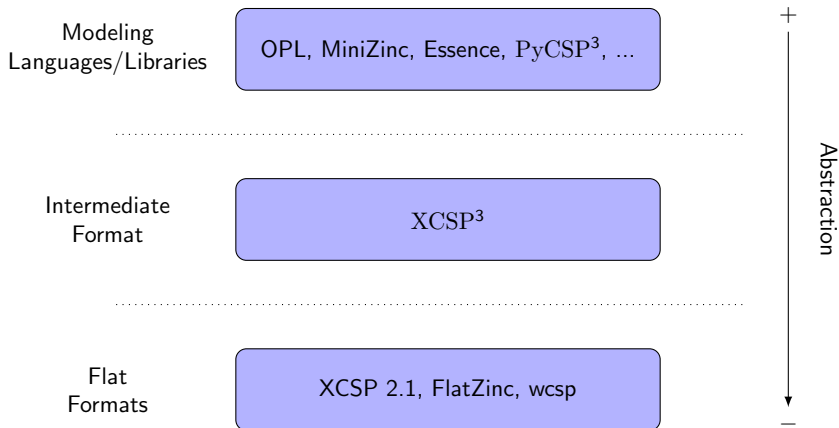
Illustrative data in: <https://www.cril.fr/~lecoutre/data/opl-example.json>



Modeling/Solving in just 20'

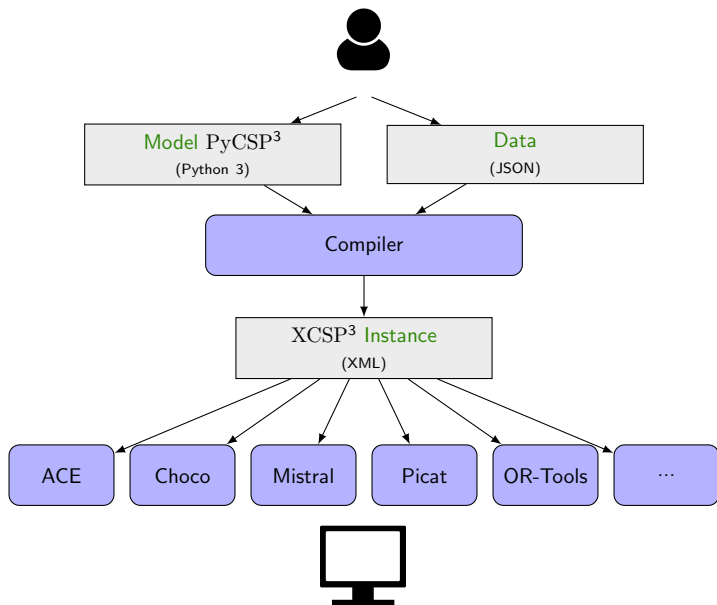
Let us start!

Modeling Languages and Formats



www.xcsp.org

A Complete Modeling/Solving Toolchain



The complete Toolchain PyCSP³ + XCSP³ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain PyCSP³ + XCSP³ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain PyCSP³ + XCSP³ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain PyCSP³ + XCSP³ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain PyCSP³ + XCSP³ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Mainstream Technologies

The complete Toolchain $\text{PyCSP}^3 + \text{XCSP}^3$ has many advantages:

- Python, JSON, and XML are robust mainstream technologies
- specifying data with JSON guarantees a unified notation, easy to read for both humans and machines
- writing models with Python 3 avoids the user learning a new language
- representing problem instances with coarse-grained XML guarantees compactness and readability

Remark.

At the intermediate level, using JSON instead of XML is possible but has some (minor) drawbacks.

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints**
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion

Popular Constraints



Popular constraints are those that are:

- often used when modeling problems
- implemented in many solvers

Remark.

XCSP³-core contains popular constraints over integer variables, classified by families.

Popular Constraints

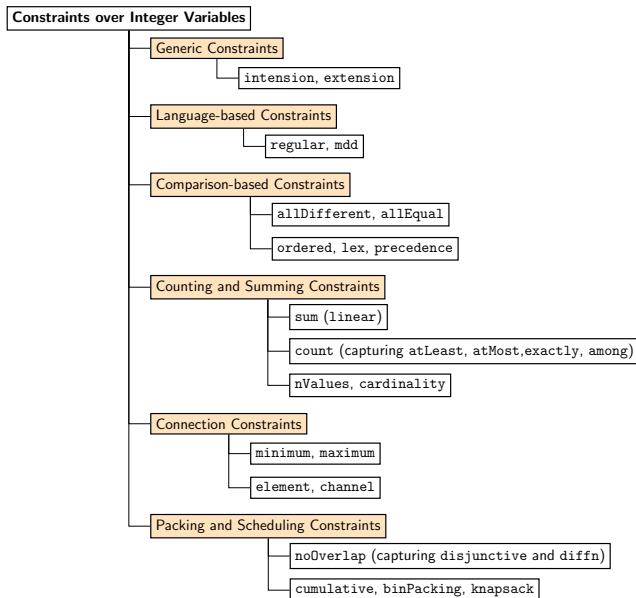


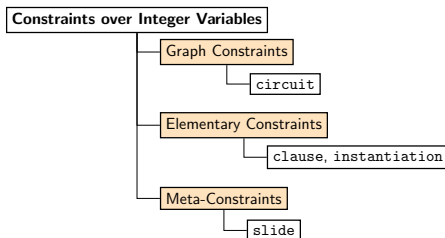
Popular constraints are those that are:

- often used when modeling problems
- implemented in many solvers

Remark.

XCSP³-core contains popular constraints over integer variables, classified by families.



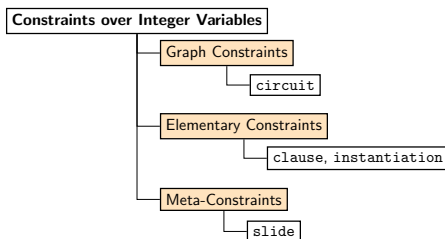


And also expressions coming from possible logical combinations of constraints in PyCSP³:

```
If (
    q < z,
    Then=h[p][y[p][q]] != h[q][y[p][q]],
    Else=y[p][q] == 0
) for p, q in combinations(k, 2)
```

Note that XCSP³-core is;

- sufficient for representing a wide range of problems (instances)
- used in XCSP³ Solver Competitions

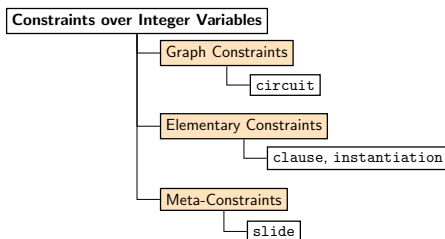


And also expressions coming from possible logical combinations of constraints in PyCSP³:

```
If (  
    q < z,  
    Then=h[p][y[p][q]] != h[q][y[p][q]],  
    Else=y[p][q] == 0  
) for p, q in combinations(k, 2)
```

Note that XCSP³-core is;

- sufficient for representing a wide range of problems (instances)
- used in XCSP³ Solver Competitions



And also expressions coming from possible logical combinations of constraints in PyCSP³:

```
If (  
    q < z,  
    Then=h[p][y[p][q]] != h[q][y[p][q]],  
    Else=y[p][q] == 0  
) for p, q in combinations(k, 2)
```

Note that XCSP³-core is;

- sufficient for representing a wide range of problems (instances)
- used in XCSP³ Solver Competitions

Generic Constraint intension

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z/2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.

Above, the examples are given in “pure” mathematical forms. For PyCSP³, operators are those of Python.

Generic Constraint intension

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z / 2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.

Above, the examples are given in “pure” mathematical forms. For PyCSP³, operators are those of Python.

Any constraint given by a Boolean expression (predicate) built from:

- variables,
- constants (integers),
- arithmetic, relational, set and logical operators.

Example.

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z / 2 = 5$
- $x + y > 3 \vee x * z = w$

Remark.

Above, the examples are given in “pure” mathematical forms. For PyCSP³, operators are those of Python.

Operators used by PyCSP³

Arithmetic Operators

+	addition
-	subtraction
*	multiplication
//	integer division
%	remainder
**	power

Relational Operators

<	Less than
<=	Less than or equal
>=	Greater than or equal
>	Greater than
!=	Different from
==	Equal to

Set Operators

in	membership
not in	non membership

Logical Operators

~	logical not
	logical or
&	logical and
^	logical xor

Illustration

Mathematical forms:

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z / 2 = 5$
- $x + y > 3 \vee x * z = w$
- $x + y > 3 \vee x * z = w$

PyCSP³ forms:

- $x > 2$
- $x \leq y + 1$
- $abs(x - y) == z - w$
- $x + y * 12 + z // 2 == 5$
- $(x + y > 3) | (x * z == w)$
- $either(x + y > 3, x * z == w)$

When compiling from PyCSP³ to XCSP³, we obtain functional forms:

```
<intension> gt(x,2) </intension>
<intension> le(x,add(y,1)) </intension>
<intension> eq(dist(x,y),sub(z,w)) </intension>
<intension> eq(add(x,mul(y,12),div(z,2)),5) </intension>
<intension> or(gt(add(x,y),3),eq(mul(x,z),w)) </intension>
```

Illustration

Mathematical forms:

- $x > 2$
- $x \leq y + 1$
- $|x - y| = z - w$
- $x + y * 12 + z / 2 = 5$
- $x + y > 3 \vee x * z = w$
- $x + y > 3 \vee x * z = w$

PyCSP³ forms:

- $x > 2$
- $x \leq y + 1$
- $abs(x - y) == z - w$
- $x + y * 12 + z // 2 == 5$
- $(x + y > 3) | (x * z == w)$
- $either(x + y > 3, x * z == w)$

When compiling from PyCSP³ to XCSP³, we obtain functional forms:

```
<intension> gt(x,2) </intension>
<intension> le(x,add(y,1)) </intension>
<intension> eq(dist(x,y),sub(z,w)) </intension>
<intension> eq(add(x,mul(y,12),div(z,2)),5) </intension>
<intension> or(gt(add(x,y),3),eq(mul(x,z),w)) </intension>
```

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

With X a sequence of variables and T a set of tuples,

- $X \in T$ is a **positive table** constraint,
- $X \notin T$ is a **negative table** constraint.

Remark.

Tuples are respectively called supports and conflicts in positive and negative tables.

We can build:

- *ordinary* tables that contain ordinary tuples
- *short* tables that contain short tuples, i.e., tuples involving the symbol '*'
- and even *compressed* tables, and *smart* tables (not in XCSP³-core)

The table constraint:

x	y	z
0	0	0
0	0	1
0	0	2
1	1	1
1	2	2
2	2	0

is written in PyCSP³ as:

```
(x,y,z) in {(0,0,0),(0,0,1),(0,0,2),(1,1,1),(1,2,2),(2,2,0)}
```

Generic Constraint extension

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	*
1	1	1
1	2	2
2	2	0

which gives in PyCSP³:

```
(x,y,z) in {(0,0,ANY),(1,1,1),(1,2,2),(2,2,0)}
```

and gives in XCSP³:

```
<extension>  
  <list> x y z </list>  
  <supports> (0,0,*) (1,1,1) (1,2,2) (2,2,0) </supports>  
</extension>
```

Generic Constraint extension

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	*
1	1	1
1	2	2
2	2	0

which gives in PyCSP³:

```
(x,y,z) in {(0,0,ANY),(1,1,1),(1,2,2),(2,2,0)}
```

and gives in XCSP³:

```
<extension>  
  <list> x y z </list>  
  <supports> (0,0,*) (1,1,1) (1,2,2) (2,2,0) </supports>  
</extension>
```

Generic Constraint extension

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	*
1	1	1
1	2	2
2	2	0

which gives in PyCSP³:

```
(x,y,z) in {(0,0,ANY),(1,1,1),(1,2,2),(2,2,0)}
```

and gives in XCSP³:

```
<extension>  
  <list> x y z </list>  
  <supports> (0,0,*) (1,1,1) (1,2,2) (2,2,0) </supports>  
</extension>
```

Generic Constraint extension

If the domain of the variable z is $\{0, 1, 2\}$, can we compress?

x	y	z
0	0	*
1	1	1
1	2	2
2	2	0

which gives in PyCSP³:

```
(x,y,z) in {(0,0,ANY),(1,1,1),(1,2,2),(2,2,0)}
```

and gives in XCSP³:

```
<extension>  
  <list> x y z </list>  
  <supports> (0,0,*) (1,1,1) (1,2,2) (2,2,0) </supports>  
</extension>
```

Understanding Constraints

For each type of constraints (around 20 basic ones):

- a dedicated Jupyter notebook
- some Jupyter notebooks building models (step by step) involving it

For example:

- <https://pycsp.org/documentation/constraints/Cumulative>
- <https://pycsp.org/documentation/constraints/Cardinality>

Understanding Constraints

For each type of constraints (around 20 basic ones):

- a dedicated Jupyter notebook
- some Jupyter notebooks building models (step by step) involving it

For example:

- <https://pycsp.org/documentation/constraints/Cumulative>
- <https://pycsp.org/documentation/constraints/Cardinality>

Illustration: Champions League

Matchday 1



...

Matchday 2



...

Matchday 3



...

...

L'EQUIPE 21

As a journalist, you can exhibit some crazy scenarii:

- What is the highest number of points that can be reached by a team without being qualified
 - in the first 8 teams
 - in the first 24 teams
- What is the lowest number of points that can be reached by a team while being qualified
 - in the first 8 teams
 - in the first 24 teams
- ...

Champions League in Practice

Tournament data in: <https://www.cril.fr/~lecoutre/data/2024.json>



Modeling/Solving in just 30'

Let us start!

```

schedule, position = data # position is 9 or 25 for example

nWeeks, nMatchesPerWeek, nTeams = len(schedule), len(schedule[0]), len(schedule[0]) * 2
assert nWeeks == 8 and nTeams == 36 # for the moment

WON, DRAWN, LOST = results = range(3)

# x[w][k] is the result of the kth match in the wth week
x = VarArray(size=[nWeeks, nMatchesPerWeek], dom=results)

# y[w][i] is the number of points won by the ith team in the wth week
y = VarArray(size=[nWeeks, nTeams], dom={0, 1, 3})

# z[i] is the number of points won by the ith team
z = VarArray(size=nTeams, dom=range(3 * nWeeks + 1))

# the target team that must be ranked at the specified position
target = Var(dom=range(nTeams))

# the number of teams with a score better than the target team
better_target = Var(dom=range(nTeams))

# the number of teams with a score equal to the target team
equal_target = Var(dom=range(nTeams))

satisfy(
    # computing points won for every match
    [
        (x[w][k], y[w][i], y[w][j]) in {(WON, 3, 0), (DRAWN, 1, 1), (LOST, 0, 3)}
        for w in range(nWeeks) for k, (i, j) in enumerate(schedule[w])
    ],

    # computing the number of points of each team
    [z[i] == Sum(y[:, i]) for i in range(nTeams)],

    ... TODO
)

maximize(
    # maximizing the number of points of the target team
    z[target]
)

```

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation**
- 6 Backtrack Search
- 7 Conclusion

Filtering Domains by means of Constraints

Each constraint represents a “sub-problem” from which some *inconsistent* values can be deleted.

Inconsistent values belong to no solution (of the sub-problem).

Several levels/types of filtering can be defined. The most popular are:

- AC (Arc Consistency): all inconsistent values are identified and deleted
- BC (Bounds Consistency): inconsistent values corresponding to the bounds of the domains are identified and deleted

Warning.

For non-binary constraints, AC is often denoted by GAC (but not in this presentation).

Filtering Domains by means of Constraints

Each constraint represents a “sub-problem” from which some *inconsistent* values can be deleted.

Inconsistent values belong to no solution (of the sub-problem).

Several levels/types of filtering can be defined. The most popular are:

- AC (Arc Consistency): all inconsistent values are identified and deleted
- BC (Bounds Consistency): inconsistent values corresponding to the bounds of the domains are identified and deleted

Warning.

For non-binary constraints, AC is often denoted by GAC (but not in this presentation).

Filtering Domains by means of Constraints

Each constraint represents a “sub-problem” from which some *inconsistent* values can be deleted.

Inconsistent values belong to no solution (of the sub-problem).

Several levels/types of filtering can be defined. The most popular are:

- AC (**Arc Consistency**): all inconsistent values are identified and deleted
- BC (**Bounds Consistency**): inconsistent values corresponding to the bounds of the domains are identified and deleted

Warning.

For non-binary constraints, AC is often denoted by GAC (but not in this presentation).

Filtering Domains by means of Constraints

Each constraint represents a “sub-problem” from which some *inconsistent* values can be deleted.

Inconsistent values belong to no solution (of the sub-problem).

Several levels/types of filtering can be defined. The most popular are:

- AC (**Arc Consistency**): all inconsistent values are identified and deleted
- BC (**Bounds Consistency**): inconsistent values corresponding to the bounds of the domains are identified and deleted

Warning.

For non-binary constraints, AC is often denoted by GAC (but not in this presentation).

Example.

Constraint $x < y$ with

- $dom(x) = 10..20$
- $dom(y) = 0..15$

After AC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

After BC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

Constraint $w + 3 = z$ with

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

After AC filtering, we obtain:

- $dom(w) = \{1, 5\}$
- $dom(z) = \{4, 8\}$

After BC filtering, we obtain:

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

Example.

Constraint $x < y$ with

- $dom(x) = 10..20$
- $dom(y) = 0..15$

After AC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

After BC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

Constraint $w + 3 = z$ with

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

After AC filtering, we obtain:

- $dom(w) = \{1, 5\}$
- $dom(z) = \{4, 8\}$

After BC filtering, we obtain:

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

Example.

Constraint $x < y$ with

- $dom(x) = 10..20$
- $dom(y) = 0..15$

After AC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

After BC filtering, we obtain:

- $dom(x) = 10..14$
- $dom(y) = 11..15$

Constraint $w + 3 = z$ with

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

After AC filtering, we obtain:

- $dom(w) = \{1, 5\}$
- $dom(z) = \{4, 8\}$

After BC filtering, we obtain:

- $dom(w) = \{1, 3, 4, 5\}$
- $dom(z) = \{4, 5, 8\}$

Generic AC Algorithm

Definition

An AC algorithm for a constraint c is an algorithm that removes all values that are arc-inconsistent on c (i.e., values that never appear in solutions of c); the algorithm is said to enforce/establish AC on c .

Here is an AC algorithm that can be used in theory with any constraint c .

Algorithm 1: filterAC(c : Constraint)

```
for each variable  $x \in scp(c)$  do
  for each value  $a \in dom(x)$  do
    if  $\neg seekSupport(c, x, a)$  // function to be implemented
    then
      remove  $a$  from  $dom(x)$ 
```

Generic AC Algorithm

Definition

An AC algorithm for a constraint c is an algorithm that removes all values that are arc-inconsistent on c (i.e., values that never appear in solutions of c); the algorithm is said to enforce/establish AC on c .

Here is an AC algorithm that can be used in theory with any constraint c .

Algorithm 2: filterAC(c : Constraint)

```
for each variable  $x \in scp(c)$  do
  for each value  $a \in dom(x)$  do
    if  $\neg seekSupport(c, x, a)$  // function to be implemented
    then
      remove  $a$  from  $dom(x)$ 
```

Generic AC Algorithm

Definition

An AC algorithm for a constraint c is an algorithm that removes all values that are arc-inconsistent on c (i.e., values that never appear in solutions of c); the algorithm is said to enforce/establish AC on c .

Here is an AC algorithm that can be used in theory with any constraint c .

Algorithm 3: filterAC(c : Constraint)

```
for each variable  $x \in scp(c)$  do
  for each value  $a \in dom(x)$  do
    if  $\neg seekSupport(c, x, a)$  // function to be implemented
    then
      remove  $a$  from  $dom(x)$ 
```

AC Filtering for allDifferent

Proposition

A constraint $\text{allDifferent}(X)$ is AC iff $\forall X' \subseteq X$,

$$|\text{dom}(X')| = |X'| \Rightarrow \forall x \in X \setminus X', \text{dom}(x) = \text{dom}(x) \setminus \text{dom}(X')$$

where $\text{dom}(X') = \cup_{x' \in X'} \text{dom}(x')$

Remark.

A subset X' of variables such that $|\text{dom}(X')| = |X'|$ is called a Hall set.

Example.

The set of variables $\{x, y, z\}$ such that:

- $\text{dom}(x) = \{a, b\}$,
- $\text{dom}(y) = \{a, c\}$
- and $\text{dom}(z) = \{b, c\}$

is a Hall set (of size 3).

AC Filtering for allDifferent

Proposition

A constraint `allDifferent(X)` is AC iff $\forall X' \subseteq X$,

$$|dom(X')| = |X'| \Rightarrow \forall x \in X \setminus X', dom(x) = dom(x) \setminus dom(X')$$

where $dom(X') = \cup_{x' \in X'} dom(x')$

Remark.

A subset X' of variables such that $|dom(X')| = |X'|$ is called a Hall set.

Example.

The set of variables $\{x, y, z\}$ such that:

- $dom(x) = \{a, b\}$,
- $dom(y) = \{a, c\}$
- and $dom(z) = \{b, c\}$

is a Hall set (of size 3).

AC Filtering for allDifferent

Proposition

A constraint $\text{allDifferent}(X)$ is AC iff $\forall X' \subseteq X$,

$$|dom(X')| = |X'| \Rightarrow \forall x \in X \setminus X', dom(x) = dom(x) \setminus dom(X')$$

where $dom(X') = \cup_{x' \in X'} dom(x')$

Remark.

A subset X' of variables such that $|dom(X')| = |X'|$ is called a Hall set.

Example.

The set of variables $\{x, y, z\}$ such that:

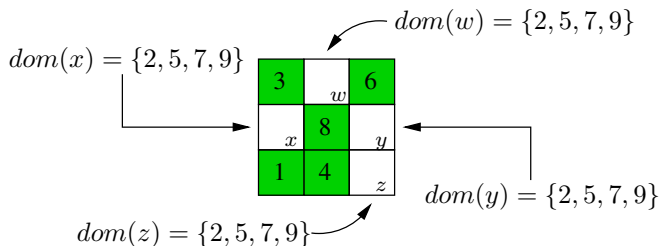
- $dom(x) = \{a, b\}$,
- $dom(y) = \{a, c\}$
- and $dom(z) = \{b, c\}$

is a Hall set (of size 3).

AC Filtering for allDifferent

Example.

For a Sudoku block, a constraint $\text{allDifferent}(w, x, y, z)$:

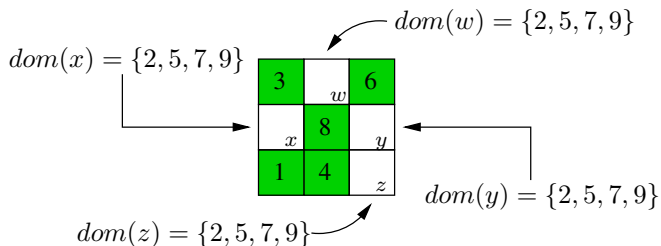


Can we filter?

AC Filtering for allDifferent

Example.

For a Sudoku block, a constraint $\text{allDifferent}(w, x, y, z)$:



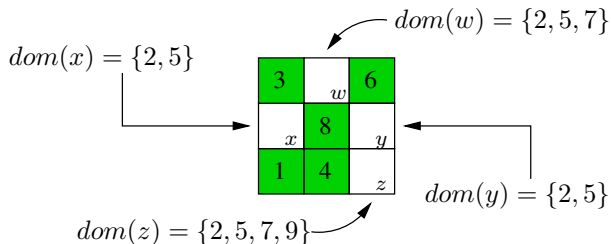
Can we filter?

Identification of Hall sets

The same constraint as previously, but variables have different domains.

Example.

For a Sudoku block, a constraint `allDifferent(w, x, y, z)` :

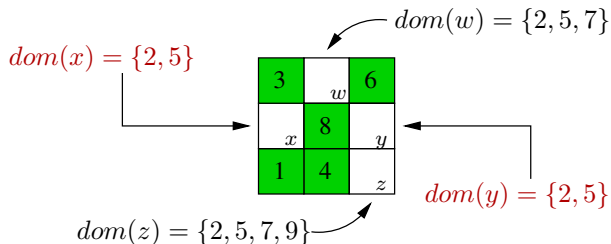


Identification of Hall sets

The same constraint as previously, but variables have different domains.

Example.

For a Sudoku block, a constraint `allDifferent(w, x, y, z)` :

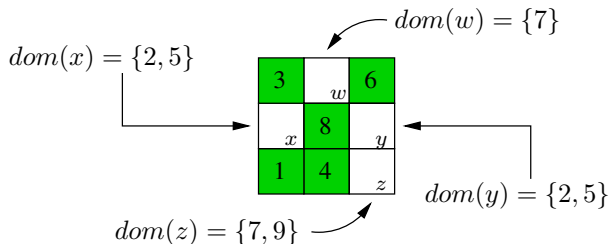


Identification of Hall sets

The same constraint as previously, but variables have different domains.

Example.

For a Sudoku block, a constraint $\text{allDifferent}(w, x, y, z)$:

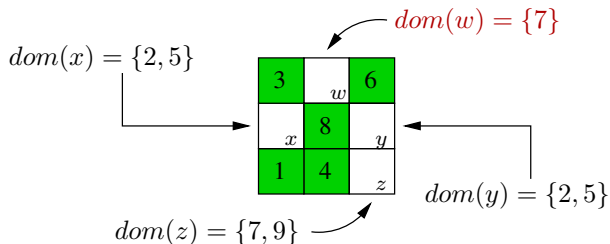


Identification of Hall sets

The same constraint as previously, but variables have different domains.

Example.

For a Sudoku block, a constraint $\text{allDifferent}(w, x, y, z)$:

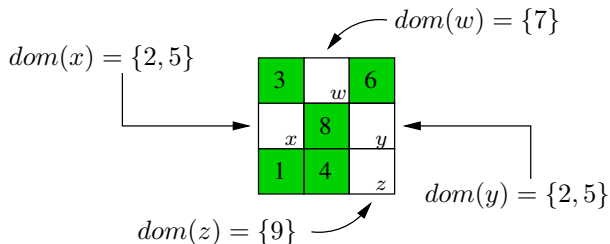


Identification of Hall sets

The same constraint as previously, but variables have different domains.

Example.

For a Sudoku block, a constraint $\text{allDifferent}(w, x, y, z)$:



Constraint Propagation

Definition

A constraint network P is AC iff each constraint of P is AC.

Definition

Computing the AC-closure of a constraint network P is the fact of removing all arc-inconsistent of P (when considering any constraint of P).

The process that involves executing filtering operations, by soliciting constraints in turn, until a fixed point is reached is called **constraint propagation**.

Constraint Propagation

Definition

A constraint network P is AC iff each constraint of P is AC.

Definition

Computing the **AC-closure** of a constraint network P is the fact of removing all arc-inconsistent of P (when considering any constraint of P).

The process that involves executing filtering operations, by soliciting constraints in turn, until a fixed point is reached is called **constraint propagation**.

Constraint Propagation

Definition

A constraint network P is AC iff each constraint of P is AC.

Definition

Computing the **AC-closure** of a constraint network P is the fact of removing all arc-inconsistent of P (when considering any constraint of P).

The process that involves executing filtering operations, by soliciting constraints in turn, until a fixed point is reached is called **constraint propagation**.

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search**
- 7 Conclusion

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^n e r)$, assuming that a constraint check is $O(r)$

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^n e r)$, assuming that a constraint check is $O(r)$

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^{ner})$, assuming that a constraint check is $O(r)$

Exponential Growth

Suppose that:

- the complexity is only $O(2^n)$
- 10^9 complete instantiations can be processed any new second



n	2^n	Processing Time
10	around 10^3	around 1 nanosecond
20	around 10^6	around 1 millisecond
30	around 10^9	around 1 second
40	around 10^{12}	around 16 minutes
50	around 10^{15}	around 11 days
60	around 10^{18}	around 32 years
70	around 10^{21}	around 317 centuries

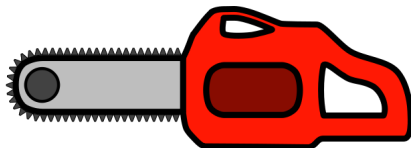
Search Tree

Most of the time, the search space can be perceived as a search tree.



Pruning the Search Tree

Constraint Inference (Filtering/Propagation) can help us!



Pruning the Search Tree

Finding a solution may become realistic in a reduced search tree.



Complete Exploration

Classical approach

- depth-first traversal
- backtracking mechanism
- interleaving of
 - decisions
 - propagations

Remark.

Other strategies exist:

- breadth-first traversal
- limited discrepancy search (LDS)
- large neighborhood search (LNS)
- ...

Complete Exploration

Classical approach

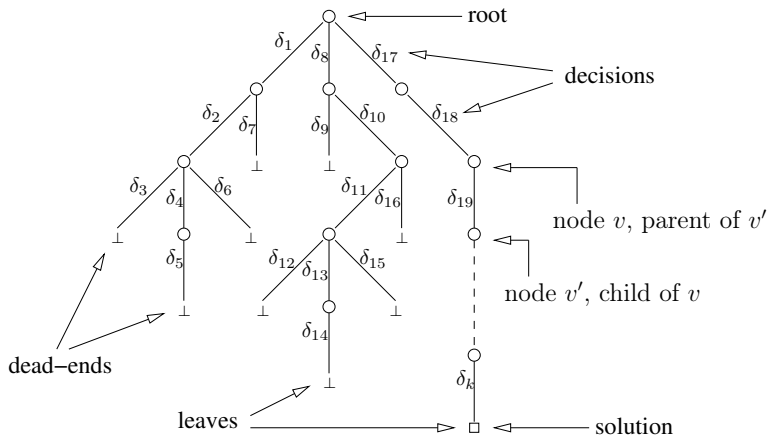
- depth-first traversal
- backtracking mechanism
- interleaving of
 - decisions
 - propagations

Remark.

Other strategies exist:

- breadth-first traversal
- limited discrepancy search (LDS)
- large neighborhood search (LNS)
- ...

Search Tree



Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to **drastically different results in terms of efficiency**.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we solicit:

- a **variable ordering heuristic** to select the next variable x to be assigned
- a **value ordering heuristic** to select the value a to assign to x

Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to **drastically different results in terms of efficiency**.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we solicit:

- a **variable ordering heuristic** to select the next variable x to be assigned
- a **value ordering heuristic** to select the value a to assign to x

Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to **drastically different results in terms of efficiency**.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we solicit:

- a **variable ordering heuristic** to select the next variable x to be assigned
- a **value ordering heuristic** to select the value a to assign to x

Search-guiding Heuristics

General rules to adopt for efficiency:

- ① It is better to assign first the variables that belong to the hard parts of the problem. **Fail-first principle:**
“To succeed, try first where you are most likely to fail”
- ② To find quickly a solution, it is better to assign first the value that most likely belongs to a solution (Succeed-first or Promise principle).
- ③ The initial variable/value choices are particularly important.

ACE, a classical CP Solver

ACE is a competitive solver because of

- restarting search frequently
- recording nogoods from restarts
- using constraint/variable weighting for selecting variables
 - $wdeg/dom$ and $wdeg^{cacd}$
 - $frba/dom$
 - $pick/dom$
- reasoning from last conflict(s)
- using solution(-based) phase saving

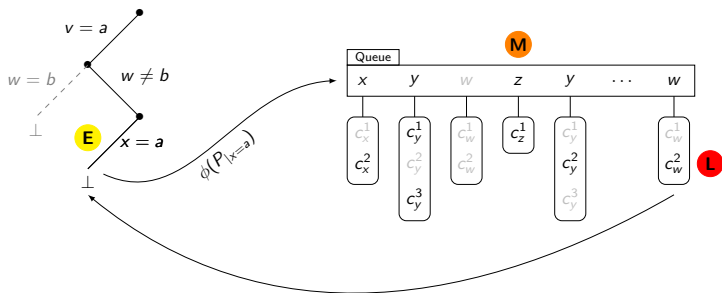


Towards Robust Search

Important issue: get complementary search mechanisms/heuristics.

Currently, we have identified three pivotal moments for collecting information about conflicts (and to guide search):

- early (E), as in *frba/dom*
- midway (M), as in *pick/dom*
- late (L), as in *wdeg/dom* and *wdeg^{cacd}*



Towards Robust Search

Showing the benefit of using complementary mechanisms/heuristics at the 2024 XCSP3 competition, by comparing two versions of ACE:

- ACE, with its default behaviour:
 - $wdeg^{cacd}$ for selecting variables
 - *first* for selecting values
- ACE-rr (mix), which exploits in a “run-robin” mode:
 - four variable ordering heuristics: *pick/dom*, *wdeg/dom*, *frba/dom*, $wdeg^{cacd}$
 - three basic value-ordering heuristics: *first*, *last*, *rand*

Of course, it is well-known that diversifying search is essential. See:

- CP-SAT
- Gurobi
- LocalSolver
- ...

Track CSP at the 2024 XCSP3 Competition

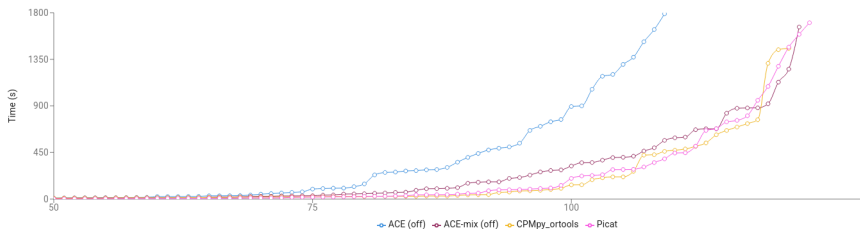
Observations:

- ACE-rr (mix) clearly outperforms ACE (default),
- ACE-rr (mix) becomes competitive with Picat and CPMpy_or-tools

Ranking of Solvers

Nb selected instances: 200

Solver	↓ #Solved	#SAT	#UNSAT	#Exclusive	#Fastest	#Unsupported
<i>Virtual Best Solver</i>	146	118	28	13	146	0
Picat	123	104	19	5	84	0
ACE-mix (off)	122	103	19	1	34	0
CPMpy_or-tools	121	98	23	6	27	0
ACE (off)	109	93	16	1	11	0



Track COP at the 2024 XCSP3 Competition

Observations:

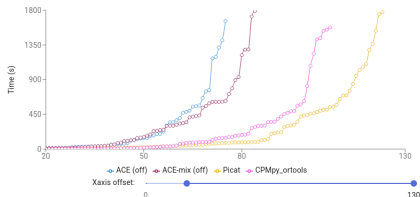
- ACE-rr (mix) clearly outperforms ACE (default) (and Picat)
- Picat and CPMpy_or-tools are far better in terms of proof
- ACE-rr (mix) is competitive with CPMpy_or-tools in terms of search

Ranking of Solvers

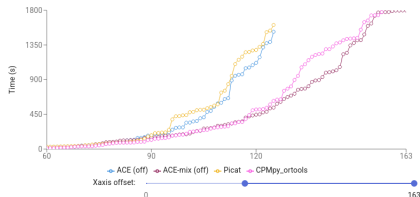
Nb selected instances: 250

Solver	↓ Score	#Opt	#BB1	#BB2	#Unsupported
Virtual Best Solver	236.00	129	97	0	0
ACE-mix (off)	157.00	84	60	26	0
CPMpy_or-tools	156.00	107	43	12	0
Picat	134.00	123	11	0	0
ACE (off)	116.50	75	27	29	0

Proof-oriented Vision



Search-oriented Vision



ACE in Practice

FAPP (<https://github.com/xcsp3team/PyCSP3-models/tree/main/realistic/FAPP>).
Compare (and also aux-02-0250):

- `java ace FAPP-aux-01-0200.xml`
- `java ace FAPP-aux-01-0200.xml -rr`
- `java ace FAPP-aux-01-0200.xml -rr -tomdd`

See also slides by Simon de Givry at CP'20 tutorials

Cargo (<https://github.com/xcsp3team/PyCSP3-models/tree/main/realistic/Cargo>).
Compare:

- `java ace Cargo-04-1s-626.xml`
- `java ace Cargo-04-1s-626.xml -varh=PickOnDom`
- `java ace Cargo-04-1s-626.xml -varh=FrbaOnDom`
- `java ace Cargo-04-1s-626.xml -rr`
- `java ace Cargo-04-1s-626.xml -varh=Dom`

Outline

- 1 Introduction to CP
- 2 CP Applications
- 3 Integrated CP Suite
- 4 Types of Constraints
- 5 Constraint Propagation
- 6 Backtrack Search
- 7 Conclusion**

Practical CP

Making CP very practical is concretized by:

- Zero installation from using Colab to start
- Easy installation ('pip install', Jupyter notebooks, etc.) to continue
- Strong intelligibility at each step of the modeling/solving process
- Complete guide with a large number of examples (<https://arxiv.org/abs/2009.00326>) ; Guide for Version 2.5 expected in August 2025
- 400 (non trivial) models/problems available: <https://pycsp.org/models/>
- Possible comparisons by analysing the results of XCSP³ competitions (<https://xcsp.org/competitions/>)



Imagine that:

- you are a **puzzle fanatic** who wants to show off on the internet (JaneStreet, LinkedIn, ...)
- you are a **journalist** at *l'equipe 21* who wants to exhibit crazy scenarii about the Champions League
- you are the **conference organizer** of ROADEF'42, and you have to build the complex programme of the conference
- you are a **music composer** who wants to deal with Tiling Rhythmic Canons, that are purely rhythmic contrapuntal compositions
- you are a **vessel captain** who has to load different cargoes to the available tanks of your vessel

CP is here for you!



Imagine that:

- you are a **puzzle fanatic** who wants to show off on the internet (JaneStreet, LinkedIn, ...)
- you are a **journalist** at *l'equipe 21* who wants to exhibit crazy scenarii about the Champions League
- you are the **conference organizer** of ROADEF'42, and you have to build the complex programme of the conference
- you are a **music composer** who wants to deal with Tiling Rhythmic Canons, that are purely rhythmic contrapuntal compositions
- you are a **vessel captain** who has to load different cargoes to the available tanks of your vessel

CP is here for you!

Last Illustration: Tank Allocation for Liquid Bulk Vessels



You have to put different cargoes (volumes of chemical products to be shipped by the vessel) to the available tanks of the vessel. You have to:

- prevent chemicals from being loaded into certain types of tanks;
- prevent some pairs of cargoes to be placed next to each other.

To minimize the inconvenience of tank cleaning, an ideal loading plan should maximize the total volume of unused tanks (i.e. free space).

Layout of the Vessel

33	29	25	21	17	13	9	5	3	1
	30	26	22	18	14	10	6		
34	31	27	23	19	15	11	7	4	2
	32	28	24	20	16	12	8		



The characteristics of this (real) instance (coming from a major chemical tanker company):

- there are 20 cargoes with volumes ranging from 381 to 1527 tons;
- the vessel has 34 tanks with capacities from 316 to 1017 tons;
- there are 5 pairs of cargoes that cannot be placed into adjacent tanks;
- each tank has between 1 to 3 cargoes that cannot be assigned to it.

Tank Allocation in Practice

Data in: <https://www.cril.fr/lecoute/data/chemical.json>



Modeling/Solving in just 15'

Let us start!

Model for Tank Allocation

```
volumes, conflicts, tanks = data
T = conflicts + [(v, u) for u, v in conflicts]
capacities, impossible_cargos, neighbours = zip(*tanks)
nCargos, nTanks = len(volumes), len(tanks)

DUMMY_CARGO = nCargos

# x[i] is the cargo (type) of the ith tank (DUMMY_CARGO, if empty)
x = VarArray(size=nTanks, dom=range(nCargos + 1))

satisfy(
    # allocating a compatible cargo to each tank
    [x[i] not in impossible_cargos[i] for i in range(nTanks)],

    # ensuring no adjacent tanks containing incompatible cargo
    [(x[i], x[j]) not in T for i in range(nTanks) for j in neighbours[i]],

    # ensuring each cargo is shipped
    [
        Sum(capacities[i] * (x[i] == cargo) for i in range(nTanks)) >= volumes[cargo]
        for cargo in range(nCargos)
    ]
)

maximize(
    # maximizing free space
    Sum(capacities[i] * (x[i] == DUMMY_CARGO) for i in range(nTanks))
)
```

- Apt, K.R. 2003.
Principles of Constraint Programming.
Cambridge University Press.
- Bessiere, C. 2006.
Constraint Propagation.
Chap. 3, pages 29–83 of: Handbook of Constraint Programming.
Elsevier.
- Dechter, R. 2003.
Constraint processing.
Morgan Kaufmann.
- Hooker, J.N. 2012.
Integrated Methods for Optimization.
Springer.
- Knuth, D. E. 2025.
The Art of Computer Programming, Volume 4, Fascicle 7:
Constraint Satisfaction.
Addison-Wesley Professional.

- Lecoutre, C. 2009.
Constraint networks: techniques and algorithms.
ISTE/Wiley.
- Régin, J.-C. 2011.
Global Constraints: a survey.
Chap. 2, pages 63–134 of: Hybrid Optimization.
Springer.
- Rossi, F., van Beek, P., & Walsh, T. (eds). 2006.
Handbook of Constraint Programming.
Elsevier.
- Tsang, E. 1993.
Foundations of constraint satisfaction.
Academic Press.
- van Hentenryck, P., & Michel, L. 2005.
Constraint-based local search.
MIT Press.