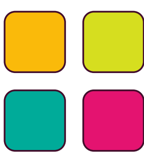


Modeling and Solving Constraint Problems

Gilles Audemard
(Christophe Lecoutre Nicolas Szczepanski)

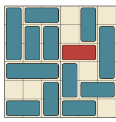
Nagoya University - April 2026

Solving Constrained Combinatorial Problems

What do you want to do ?

Solving Constrained Combinatorial Problems

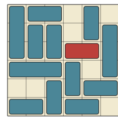
What do you want to do ?



Play puzzle games (sudoku, Jane Street. . .)

Solving Constrained Combinatorial Problems

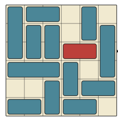
What do you want to do ?



Find the worst/best case for your favorite team during the initial phase

Solving Constrained Combinatorial Problems

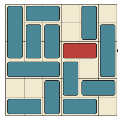
What do you want to do ?

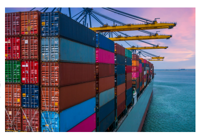


build the complex programme of IJCAI'2030 conference

Solving Constrained Combinatorial Problems

What do you want to do ?

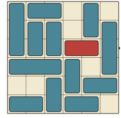


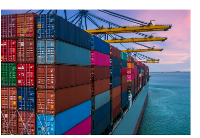


Maximize the number of items to transport

Solving Constrained Combinatorial Problems

What do you want to do ?

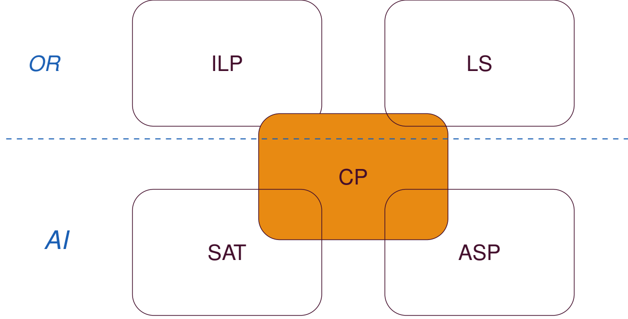




In all cases, you need to solve combinatorial problems

Solving Constrained Combinatorial Problems

- Use a dedicated algorithms for the initial problem.
- Use a formalism and specialized solvers available.

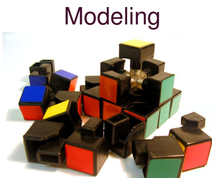


Each paradigm has its strengths and weaknesses

Constraint Programming

Two main stages in Constraint Programming (CP)

Modeling



Solving

- The solving part is hard, but algorithms exist and can be easily compared.
- The modeling part seems easy, but. . . it is not really the case.

The Need for a Common Representation Language

The SAT case : DIMACS format (1993)

- The input language is simple.
- Helps to compare solvers on same instances.
- Helps to propose competitions.
- Promotes the SAT technology.

The Need for a Common Representation Language

The SAT case : DIMACS format (1993)

- The input language is simple.
- Helps to compare solvers on same instances.
- Helps to propose competitions.
- Promotes the SAT technology.

The C(S)P case :

- A huge number of constraints (with plenty of variants)
- Harder to represent
- Different frameworks proposed over the years
  - Essence : elegant (mathematical foundations) but requires some 'expertise' in Mathematics
  - Minizinc : rather widespread, but uneasy syntax and flattened heavy intermediate files
- Sophisticated toolchains
- But not suited for a wide audience : too verbose, too obfuscated

Our contributions

XCSP^3 and PyCSP^3

- XCSP^3, a universal format for representing constraint problems (based on XML).

  https://xcsp.org

- PyCSP^3, a Python library for modeling Constraint Problem.

  https://pycsp.org

- Available tools (parsers, checkers).

  https://github.com/xcsp3team/

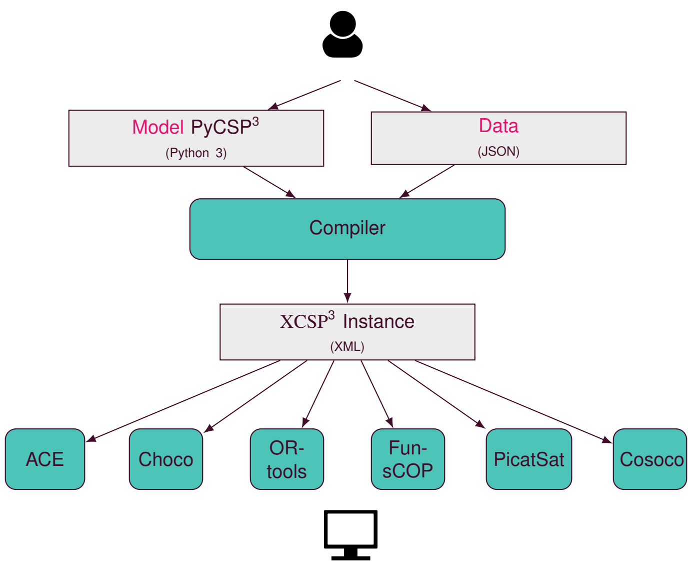
- More than 400 models available.

  https://github.com/xcsp3team/pycsp3-models

- A competition organized each year.

  https://www.cril.univ-artois.fr/XCSP25/

Global Architecture

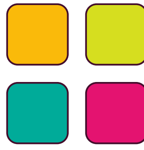


Interest of this approach

- Fast modeling with a dedicated Python interface (not a new language to learn).
- An easy way to process data.
- Intermediate format in XML.
  - Readable/editable by humans (remove some parts, understand the current model).
  - Preserve the structure of models.
- Integrated solvers (Choco, ACE, OrTools/CP-SAT (forthcoming)).

Intelligibility at every stage of the modelling and solving process

Let us model some problems to check this



Modeling

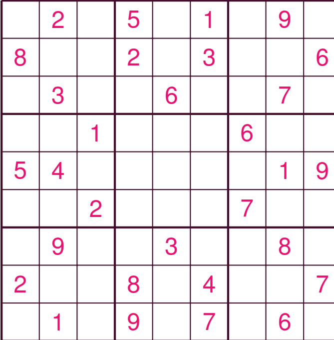
Modeling Languages

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

1. the identification of the parameters, i.e., the structure of the data
2. the description of the model, taking the parameters into account, and using an appropriate language
3. the generation of the effective data, each time a new instance has to be handled

Let us illustrate this with the classical Sudoku Problem.

Ths Sudoku Problem



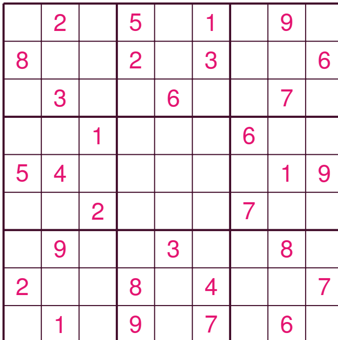
Fill the empty cells with values in 1..9

While considering that :

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

Ths Sudoku Problem

Sudoku



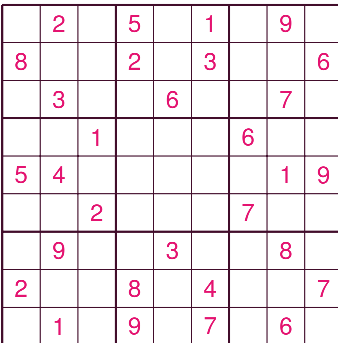
Fill the empty cells with values in 1..9

While considering that :

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

Ths Sudoku Problem

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	

Domains

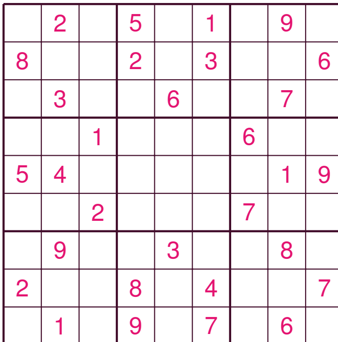
Fill the empty cells with values in 1..9

While considering that :

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

Ths Sudoku Problem

Sudoku



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

Constraints

The Sudoku Problem

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

The Sudoku Problem

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

The Sudoku Problem

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

Let us model this problem with a Jupyter Notebook

We have to :

1. identify the parameters (structure of the data) for this problem : The grid with clues.
2. exhibit variables : The empty cells with domain from 1 to 9.
3. describe a model.
4. generate effective data for different problem instances

	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	

The resulting PyCSP^3 model

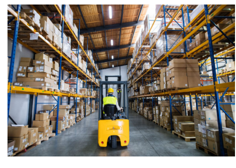
```
n = 9

x = VarArray(size=[n, n], dom=range(1, n + 1))

satisfy(
  # imposing distinct values on each row and each column
  AllDifferent(x, matrix=True),

  # imposing distinct values on each block  tag(blocks)
  [AllDifferent(x[i:i + base, j:j + base]) for i in range(0, n, base) for j in range(0, n,
     ➥base)]

  # Imposing clues
  [x[i,j] == clues[i][j]  for i in range(n) for j in range(n) if clues[i][j] > 0]
)
```

```
python3 Sudoku.py -data=grid01.json
```

XCSP3 Instance : Sudoku-grid01

```
<instance format="XCSP3" type="CSP">
  <variables>
    <array id="x" note="x[i][j] is the value of cell with coordinates (i,j)" size="[9][9]">
     ➥1..9 </array>
  </variables>
  <constraints>
    <allDifferent note="imposing distinct values on each row and each column">
      <matrix> x[][] </matrix>
    </allDifferent>
    <group class="blocks" note="imposing distinct values on each block">
      <allDifferent> %... </allDifferent>
      <args> x[0..2][0..2] </args>
      <args> x[0..2][3..5] </args>
      <args> x[0..2][6..8] </args>
      <args> x[3..5][0..2] </args>
      <args> x[3..5][3..5] </args>
      <args> x[3..5][6..8] </args>
      <args> x[6..8][0..2] </args>
      <args> x[6..8][3..5] </args>
      <args> x[6..8][6..8] </args>
    </group>
    <instantiation note="imposing clues tag">
      <list> x[0][1] x[0][3] x[0][5] x[0][7] x[1][0] x[1][3] x[1][5] x[1][8] x[2][1] x[2][4] x
     ➥[2][7] x[3][2] x[3][6] x[4][0..1] x[4][7..8] x[5][2] x[5][6] x[6][1] x[6][4] x[6][7] x
     ➥[7][0] x[7][3] x[7][5] x[7][8] x[8][1] x[8][3] x[8][5] x[8][7] </list>
      <values> 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 </values>
    </instantiation>
  </constraints>
</instance>
```

The Warehouse Problem



- A company considers opening warehouses at some candidate locations in order to supply its existing stores.
- Each possible warehouse has the same maintenance cost.
- Each warehouse has a capacity which indicates the maximum number of stores that it can supply.
- Each store must be supplied by exactly one open warehouse.
- The supply cost to a store depends on the warehouse.

Minimize the sum of the maintenance and supply costs

A Model for the WareHouse Problem

Model File `WareHouse.py`

```
from pycsp3 import *

fixed_cost, capacities, costs = data  # fixed_cost is the fixed cost when opening a
      ➥warehouse
nWarehouses, nStores = len(capacities), len(costs)

# w[i] is the warehouse supplying the ith store
w = VarArray(size=nStores, dom=range(nWarehouses))

satisfy(
    # capacities of warehouses must not be exceeded
    Count(w, value=j) <= capacities[j] for j in range(nWarehouses)
)

minimize(
    # minimizing the overall cost
    Sum(costs[i][w[i]] for i in range(nStores)) + NValues(w) * fixed_cost
)
```

```
python3 Warehouse.py -data=opl.json
```

XCSP3 Instance : Warehouse-opl

Tank Allocation for Liquid Bulk Vessels Problem

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.

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))
)
```

The resulting XCSP^3 model

```
python3 Sudoku.py -data=grid01.json
```

- See the resulting file

The Traveling Tournament Problem (TTP)

- Each team plays twice every other team (at home 🏠 and away 🌍)
- For n teams : $2 * (n - 1)$ rounds of $n/2$ games
- For each game :
  - one team is at home 🏠
  - one team is away 🌍

Round 1	Round 2	Round 3
Paris vs Lille Marseille vs Nice	Paris vs Marseille Lille vs Nice	Paris vs Nice Lille vs Marseille

Round 4	Round 5	Round 6
Lille vs Paris Nice vs Marseille	Marseille vs Paris Nice vs Lille	Nice vs Paris Marseille vs Lille

The Traveling Tournament Problem (TTP)

- Distances between team sites are given by a symmetric matrix :

	Paris	Lille	Marseille	Nice
Paris	0	200	775	931
Lille	200	0	1000	1157
Marseille	775	1000	0	200
Nice	931	1157	200	0

- Consecutive sites for a team constitute a road trip :

			Marseille		Nice		
Paris	🏠 0	→ →	🌍 775	→ →	🌍 200	→ →	🏠 931

Some constraints :

- No team can play consecutively more than two times at home and away
- No two consecutive games can involve the same pair of teams
- ...

Objective : minimizing the total distance travelled by all teams

PyCSP^3 Model for TTP

Data File `inst01.json`

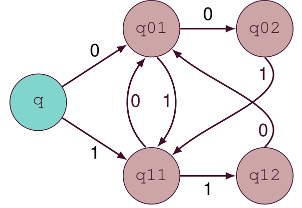
```
{
  "distances":[[0,200,775,931],[200,0,1000,1157],[775,1000,0,200],[931,1157,200,0]]
}
```

Model File `TTP.py`

```
# Data
distances = data.distances
nTeams = len(distances)
nRounds = 2 * (nTeams - 1)

# o[i][k] is the opponent (team) of the ith team  at the kth round
o = VarArray(size=[nTeams, nRounds], dom=range(nTeams))

# h[i][k] is 1 iff the ith team plays at home at the kth round
h = VarArray(size=[nTeams, nRounds], dom={0, 1})

# t[i][k] is the travelled distance by the ith team at the kth round.
t = VarArray(size=[nTeams, nRounds + 1], dom={d for row in distances for d in row})


minimize(
    # minimizing summed up travelled distance
    Sum(t)
)
```

Constraints *regular* for TTP

- No team plays more than two consecutive home/away games
- Global constraint regular [Pesant, 2004]




```
def automaton():
  t=[("q",0,"q01"),("q",1,"q11"),("q01",0,"q02"),("q01",1,"q11"),
     ("q11",0,"q01"),("q11",1,"q12"),("q02",1,"q11"),("q12",0,"q01")]
  return Automaton(start="q",final=["q01","q02","q11","q12"],transitions=t)

satisfy(
     ...
    # at most 2 consecutive games at home, or consecutive games away
    [h[i] in automaton for i in range(nTeams)],
    ...
)
```

Table Constraints for TTP

For computing travelled distances in our model

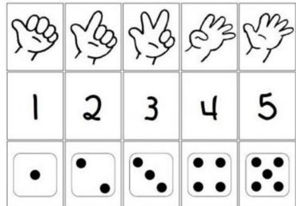
h[Paris][0]	h[Paris][1]	o[Paris][0]	o[Paris][1]	t[Paris][1]
1	1	*	*	0
0	1	Lille	*	200
0	1	Marseille	*	775
0	1	Nice	*	931
1	0	*	Lille	200
1	0	*	Marseille	775
1	0	*	Nice	931
0	0	Lille	Marseille	1000
0	0	Lille	Nice	1157
0	0	Marseille	Nice	200
		...		

```
def table(i):
    t = [(1, 1, ANY, ANY, 0)]
    t += [(0, 1, j, ANY, distances[i][j]) for j in range(nTeams) if j != i]
    t += [(1, 0, ANY, j, distances[i][j]) for j in range(nTeams) if j != i]
    t += [(0, 0, j1, j2, distances[j1][j2]) for j1 in range(nTeams) for j2 in range(nTeams)
      ➥if different_values(i, j1, j2)]
    return t

satisfy(
       ...
       # handling travelling for two successive games
       [(h[i][k], h[i][k + 1], o[i][k], o[i][k + 1], t[i][k + 1]) in table(i) for i in range
      ➥(nTeams) for k in range(nRounds - 1)],
       ...
)
```

Constraints *cardinality* for TTP

Each team must play exactly two times against each opponent

```
satisfy(
    ...
    # each team must play exactly two times against each other team
    [Cardinality(o[i], occurrences={j: 2 for j in range(nTeams) if j != i}) for i in range(
      ➥nTeams)],
    ...
)
```



The resulting model

https://github.com/xcsp3team/PyCSP3-models/tree/main/recreational/TravelingTournament

Popular Constraints

- Often used when modeling problems.
- Implemented in many solvers.
- Sufficient for representing a wide range of problems.
- Used in XCSP^3 competitions.
- For each constraint a dedicated notebook explaining it.

Type	Constraints
Generic	intension, extension
Langage-based	regular, MDD
Comparison-based	allDifferent, allEqual, ordered, lex, precedence
Counting / Summing	sum, count, nValues, cardinality
Connection	minimum, maximum, element, channel
Packing / Scheduling	noOverlap, cumulative, binPacking, knapsack
Graph	circuit

Why Python ?

Python for modeling a problem has numerous advantages :

- easy to learn and use
- natural data and control structures
- the most popular language (soon)

This is why Python is used in many libraries :

- in computer science : PyML, Pygame, TensorFlow. . .
- in many other domains : NumPy, SciPy, . . .

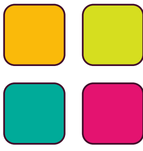
Data structures are very useful and natural !

Modeling : a (Very) Important Step

- SAT Parity problem
- Challenging instances during many years (one of the 10 SAT challenges stated in 1997 by Selman)
- Specialized techniques were introduced to solve it.

Olivier Bailleux, Yacine Boufkhad : Efficient CNF Encoding of Boolean Cardinality Constraints. CP 2003 : 108-122

- The problem was not so difficult, but the encoding was not the good one !



Solving

Generate and Test

- For a given Constraint Staisfaction Problem 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 ? $O(d^n er)$

Generate and Test

- For a given Constraint Staisfaction Problem 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 ? $O(d^n er)$

10^9 instructions per second

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

One needs to reduce the search space

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

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

Constraint $x < y$

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

After AC filtering, we obtain :

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

After BC filtering, we obtain :

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

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

Constraint $x < y$

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

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 $w + 3 = z$

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

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 `seekSupport(c,x,a)` == false **then**
      remove a from $dom(x)$

- Need to implement the function `seekSupport` for all constraints.

AC Filtering for `allDifferent`

- If a variable x_1 has only one value. Remove this value in other variables.

AC Filtering for `allDifferent`

- If a variable x_1 has only one value. Remove this value in other variables.
- If the union of domain of x_1 and x_2 is 2. These two values are mandatory for x_i and x_j. Remove them in other variables.

AC Filtering for `allDifferent`

- If a variable x_1 has only one value. Remove this value in other variables.
- If the union of domain of x_1 and x_2 is 2. These two values are mandatory for x_i and x_j. Remove them in other variables.
- If the union of domain of x_1, x_2 and x_3 is 3. These three values are mandatory for x_1, x_2 and x_3. Remove them in other variables.
- ...

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

AC Filtering for `allDifferent`

- If a variable x_1 has only one value. Remove this value in other variables.
- If the union of domain of x_1 and x_2 is 2. These two values are mandatory for x_i and x_j. Remove them in other variables.
- If the union of domain of x_1, x_2 and x_3 is 3. These three values are mandatory for x_1, x_2 and x_3. Remove them in other variables.
- ...

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

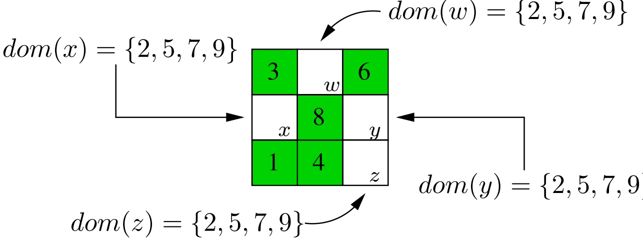
A subset X' of variables such that $|dom(X')| = |X'|$ is called a Hall set.
The set of variables $\{x, y, z\}$ such that :

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

is a Hall set (of size 3).

AC Filtering for `allDifferent`

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

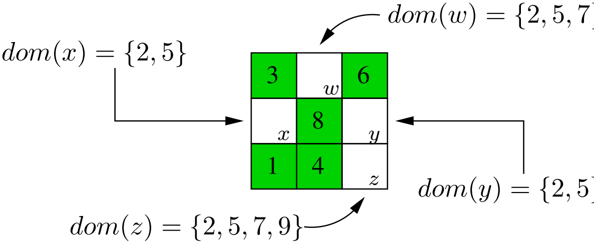


Can we filter ?

Identification of Hall sets

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

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

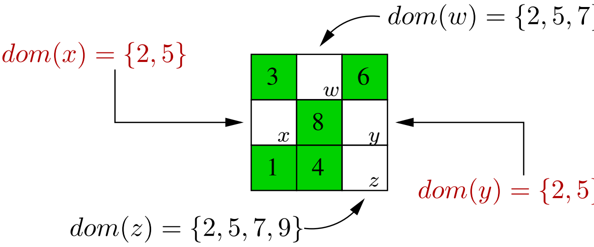


Can we filter ?

Identification of Hall sets

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

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

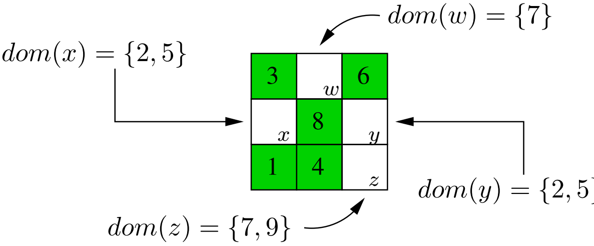


Can we filter ?

Identification of Hall sets

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

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

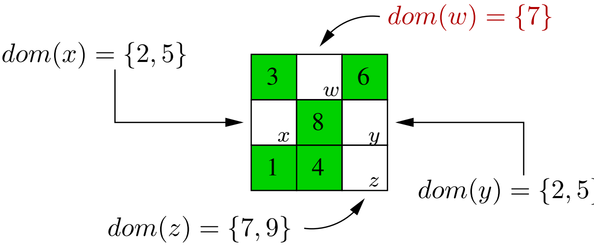


Can we filter ?

Identification of Hall sets

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

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

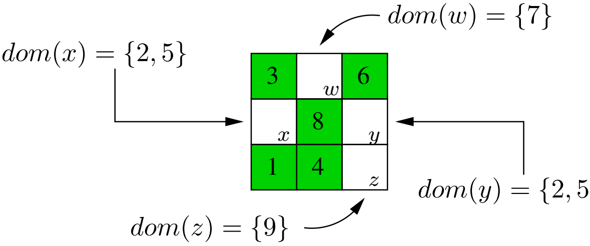


Can we filter ?

Identification of Hall sets

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

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



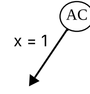
Can we filter ?

Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).

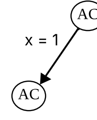
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



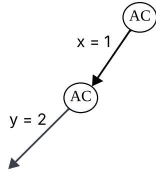
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



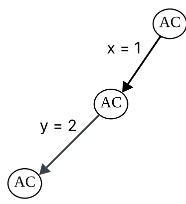
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



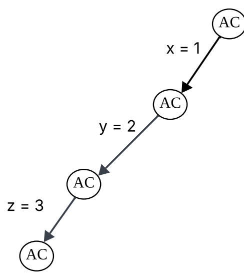
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



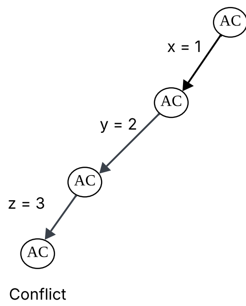
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



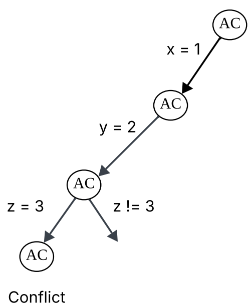
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



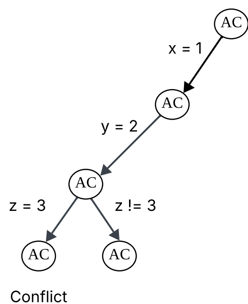
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



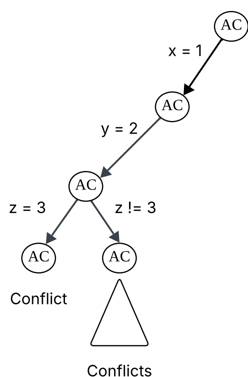
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



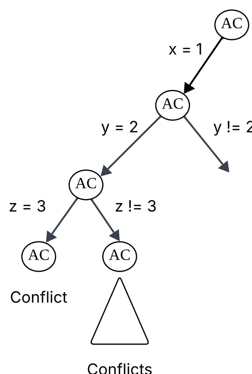
Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).



Search Tree

- Most of the time, the search space can be perceived as a search tree.
- Interleaving of decisions/propagations (AC/BC, Weak AC...)
- On backtrack, remove the conflicting value from the variable and continue the search (binary search tree).

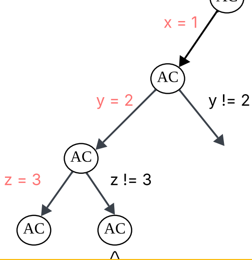


Choosing Variables and Values

Using different search ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency

- Variable heuristic : It is better to assign first the variables that belong to the hard parts of the problem. Fail-first principle :
  "*To succed, try first where you are most likely to fail*"
  - Dynamic heuristic : each time a conflict occurs, some statistics are updated. The next variable is selected wrt such statistics (wdeg, frba, cacd, pick).
  - One prefers variable with small domains, the statistic is divided by the size of the current domain.

- Value Heuristic : to find quickly a solution, it is better to assign first the value that most likely belongs to a solution. Promise principle.



Important techniques

- Restarts
  The first decision made is crucial. Regularly, restart the search from the root to reduce such impact.
- Nogoods from restart
  A simple technique to learn some UNSAT part of the search tree at each restart.
- Progress Saving
  Store the last good value for a variable and use it if it is in the domain when assign this variable.
  Allows to capture some SAT part of the search tree.

Cosoco : my own solver (*)

- Written in C++.
- Easily understandable (no template programming).
- All constraints of XCSP3-Core.
- Satisfaction, optimization
- Heuristics : Wdeg, Pick, Frba, CACD.
- Restarts, Nogoods from restarts.
- Multi-thread possibilities (portfolio) with capacity of sharing some informations.

Part	Lines of code
Data Structures (Minisat)	1,300
Core	500
Solver	3,000
Optimizer	250
Constraints	10,000
Parsing/Manage intension	4,000
Total - with headers :(	$\approx$ 19,000

(*) With the (huge) support of Christophe Lecoutre

The trail : a la MiniSAT

`trail`

`trail_lim`

- Root propagations : x_1 and x_5
- Decision Variable at level 1 : x_2, followed by domain reduction of x_1 and x_3
- Decision variable at level 2 : x_1, followed by domain reduction of x_3

- `trail` : For each decision level, the stack of modified variables
- `trail_lim` : the size of the trail at a given level
- `trail_lim.size()` : the number of decision levels.
- Source code of `newDecision` and `backtrack`

Propagators

- Filtering is achieved until a fix point
- Currently, only one queue (postponable constraints)
- The solver is responsible of domain reduction/assignment
- During the filtering process, the constraint asks the solver to remove some values
- See source code of `DiffXY::filter`

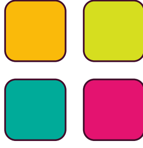
Results to XCSP25 competition

https://www.cril.univ-artois.fr/XCSP25/cop/cop

Results to XCSP25 competition

`https://www.cril.univ-artois.fr/XCSP25/cop/cop`

Some improvements since 10 months !



Conclusion

Practical CP

Make CP accessible to everybody !

- A simple modeling language based on Python.
- Easy installation (`pip install pycsp3`).
- Strong intelligibility at each step of the modeling/solving process.
- Documentation with examples.
- More than 400 models available (`https://github.com/xcsp3team/pycsp3-models`).

Tools

https://github.com/xcsp3team/

Parsers :

- Java 8 Parser
- C++ 11 Parser
- Python3 parser

Checkers

- solutions and bounds
- Instances Validity

Reproductibility

Competitions

- All instances of the XCSP competitions are available (with models and data).
- All solvers results are available (through Zenodo).
- The competition web site allows to compare solvers (while possibly using some query filters about constraints, domain size, ...) .