Problem Modeling and Solving: an Introduction to Constraint Programming

Eric Monfroy

Acknowledgement

Some slides (e.g., about applications) are largely inspired by the introduction to constraint programming of Christophe Lecoutre.

Overview

- First steps in constraint programming?
  - what is CP?
  - some applications
  - context
- Modeling
  - let's try it!
  - and it works!!!
- Constraint solving
  - filtering
  - constraint propagation
  - search

Firsts steps in CP

What is CP about?

Constraint Programming

Constraint Programming

=

modeling and solving problems under constraints

You has always known CP

World of blocks (planning and placement problem)

You has always known CP

A matching problem

"History" of constraint programming

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."

Eugene C. Freuder, Inaugural issue of the Constraints Journal, 1997.

Constraint programming paradigme

Constraint Programming paradigme

=

focus on WHAT not on HOW

⇒ the user models her/his problem, the "solver" solves it.

- solver: algorithms for computing solutions of a model instance
- solution: a set of values for variables that:
  - satisfies the constraints
  - satisfies the constraints and optimizes a criterion
  - ...

You already know constraint programming

- you can model it
- you can solve it

You can model Sudoku

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

- fill the empty cells with $\{1, \ldots, 9\}$
- such that
  - numbers of a line are different
  - numbers of a column are different
  - numbers of a 3x3 block are different

In fact: variables, domains, and constraints

Solution: assignment of values to variables satisfying the constraints

You can solve Sudoku 1/2

Removing impossible candidate values:

- if a cell is 4
  ⇒ remove 4 from the possibilities of the other cells of the row
- in a row, if 2 and 6 are the only candidates of 2 cells
  ⇒ remove 2 and 6 from the possibilities of the other cells
- . . .
- in a row, if 5 cells have the same 5 candidate values
  ⇒ these 5 values can be removed from other cells
- X-wing, XY-wing, . . .

⇒ and you iterate

these are the "filtering" and propagation techniques of constraint solvers

You can solve Sudoku 2/2

When there is no more filtering to perfom:

1. choose a cell c and fix it with one of its remaining value v
2. apply propagation again
3. if you obtain a solution
   ⇒ done
4. otherwise:
   4.1 restore c as it was before 1.
   4.2 remove v from c
   4.3 execute 1. again

⇒ and you iterate up to a solution

- 1. **is labeling/enumeration**
- 4. **is backtracking**
- **together they constitute search**

So, you do constraint programming!

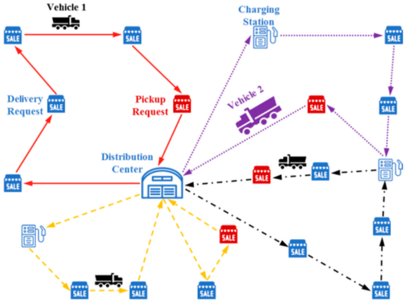
- you can model a problem
- you know how does a solver work

Some applications of CP

Early successes

- Scene labeling (Waltz 75)
- Circuit design (Siemens)
- Container port scheduling (Hong Kong and Singapore)
- Transport: SNCF, British Airways, Cathay Pacific, . . .
- Industry: Renault, British Telecom, . . .

Vehicle Routing Problem



- Given:
  - a set of customers
  - their demands
  - a fleet of trucks
  - a depot
- Find:
  - the best route for each truck
  - satisfying each client

Ship loading

- Given:
  - ships
  - containers
  - cranes
  - requests to load containers on ships
- Find:
  - the fastest loading schedule
  - meeting the requests

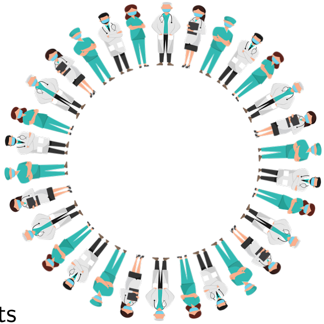
$\longleftarrow$ without CP

with CP $\longrightarrow$

Car sequencing

- Given:
  - cars to produce
  - option set for each car
  - machines installing the options
- Find:
  - the best production order
  - respecting the capacity of the machines

Nurse Rostering



- Given:
  - nurses with qualifications and shifts
  - constraints on the working/resting patterns
  - needs of the hospital for each service
- Find:
  - an assignment of shifts to nurses
  - maximizing additional criteria (equity between nurses, etc.)
  - respecting all the constraints

Some more applications

- Job shop scheduling
- Assembly line smoothing and balancing
- Cellular frequency assignment
- Shift planning
- Maintenance planning
- Airline crew rostering and scheduling
- Airport gate allocation and stand planning
- Production scheduling
- Transport scheduling (food, nuclear fuel)
- Course timetabling
- Smart cities
- Sport scheduling
- Mining problem
- Wine blending

Context

History

- Issued (mainly) from Artificial Intelligence (symbolic AI)
- First, CLP: Constraint Logic Programming (1987)
  - declarative programming
  - first systems: Prolog III, CLP, CHIP, ECLiPSe
- Then, CP: Constraint Programming
  - its own paradigme
  - creation of languages/libraries
  - creation of autonomous solvers
  - separation from the logic world

Target

Problems subject to constraints and/or objective functions

Optimization tools:

- Mathematical Programming
  - Linear Programming
  - Integer Linear Programming
  - Nonlinear programming
- Meta-heuristics
  - Trajectory-based (LS)
  - Population-based (GA, EA)
- **Constraint Programming**

Advantages of CP

- Powerful **modeling language**.
  - Simpler models (global constraints, high level languages).
  - Problem structure is kept (up to solving)
- Easy problem variant modeling
  - adding/removing some constraints
  - no need for changing solving algorithm
- Complete solvers
  - find solution if there is one
  - find optimal solution
  - prove unsatisfiability

Disadvantages of CP

- Less robust
  - does not always scale well (blow up)
  - but global constraints may help
- Less effective for continuous optimization
  - relies on interval propagation
  - but provides certified computation
- Less highly engineered softwares
  - young field and community

Constraint programming

CP: general framework with generic and efficient algorithms for solving problems under constraints

Attractive: **clear separation** between

- **modeling**: simple representation of many problems
- **solving**: many generic algorithms and heuristics to find solutions

Modeling and Solving are independent processes:

- a model can be applied various solvers or search techniques
- a solver is used for different models and problems

Modeling

A “tentative” definition

modeling
=
declaratively describe a real-life problem into a “formal” language

- the problem is generally given as some verbal statements
- the target language may generally use some mathematical or logical objects (arithmetic equations, Boolean connectors, . . .)
- the target language is preferably understandable by a solver

How to model

Typically as:

- a Constraint Satisfaction Problem (CSP)
- or a Constraint Optimization Problem (COP)

A CSP is given by:

- some (decision) variables
- some domains for variables (i.e., some candidate values)
- some relations (constraints) between variables

A COP is given by:

- a CSP
- an objective function to be optimized

CP: solving models of problems, stated as CSP/COP instances

Remarks

- a (decision) variable is a mathematical variable (not a container as in CS)
- a variable can take at most one value of its domain (if the domain is empty, the problem has no solution)
- a constraint is a "real" relation (as an equation)
  Thus:
  - $x = x + 1$ correct, but no solution (or unsatisfiable)
  - $3 = x + y$ correct constraint
  - $x = 3 \wedge x = 4$ no solution
- variables and constraints can have numerous types:
  - Integer FD with linear, quadratic, or non linear constraints
  - "real" (continuous) and linear/non linear constraint
  - Boolean variables and logic formulas (SAT)
  - set variables and set constraints
  - lists, symbolic, functions, trees, ...

Let's try it!

A simple example: the *n*-queen problem

Place n chess queens on an $n \times n$ chessboard so that no two queens threaten each other.

- do you know this problem?
- can you solve it?
- how much time do you need to solve it?

The n-queen problem

Place n chess queens on an $n \times n$ chessboard so that no two queens threaten each other.

- a verbal description of the problem
- some implicit knowledge:
  - what is a chessboard?
  - a queen can move?
  - how does a queen move?
  - threaten or "attack"? what does it mean?
  - ...

$\Rightarrow$ **let's try to clarify and model this problem**

The *n*-queen problem

nothing to optimize, thus CSP

- the manipulated "objects"
  - how to represent the chessboard?
    - a set of n^2 cells?
    - a 2-d array?
    - a set (1-d array) of columns?
  - how to represent the queens?
  - integer, Boolean domains?

  $\Rightarrow$ the variables
- relations between the "objects"
  - no 2 queens in attack?
    - not 2 queens on the same row
    - not 2 queens on the same column
    - not 2 queens on the same diagonal
  - a relation between these 2 queens?

  $\Rightarrow$ the constraints

n-queen: a first model

- Variables: $x_{i,j}, \quad i,j \in [1,n]$
  - 1: if there is a queen in row i, column j
  - 0: otherwise
- Domains (candidate values): $\{0,1\}$
- Constraints (requirements):
  - one and only one queen per row: $\forall i \in [1,n], \ \sum_{j=1}^{n} x_{i,j} = 1$
  - one and only one queen per column: $\forall j \in [1,n], \ \sum_{i=1}^{n} x_{i,j} = 1$
  - 0 or 1 queen per diagonal (4.n - 6 diagonals)
    $x_{2,1} + x_{1,2} \leqslant 1$
    $x_{3,1} + x_{2,2} + x_{1,3} \leqslant 1$
    $\ldots$

Note: "$= 1$" can be replaced by "$\leqslant 1$" adding $\sum_{i=1}^{n}(\sum_{j=1}^{n} x_{i,j}) = n$

About the first model

Model:

- a 2-d array of cells/variables
- a variable = presence or not of a queen

Family:

- Pseudo-Boolean variables and constraints
- or 0/1 Integer Linear Programming
- or integer Finite Domain (FD) model

$\Rightarrow$ **can be solved by an Integer FD constraint solver**

n-queen: a second model

- Variables and Domains: $x_{i,j} \in \{true, false\}$ for $i, j \in [1, n]$
  - true, if there is a queen at the intersection of (row i, column j)
  - false, otherwise
- Constraints:
  - one and only one queen per row:

$$\forall i \in [1, n], \quad \left(\bigvee_{j \in [1,n]} x_{i,j} \right) \bigwedge \left(\bigwedge_{j,k \in [1,n], j \neq k} (\neg x_{i,j} \vee \neg x_{i,k}) \right)$$

  - one and only one queen per column: similar, switching i and j
  - diagonals (the main diagonal only)

$$\bigwedge_{i,j \in [1,n], i \neq j} (\neg x_{i,i} \vee \neg x_{j,j})$$

About the second model

Model:

- a 2-d array of cells/variables
- a variable = presence or not of a queen

Family:

- Boolean variables and constraints

$\Rightarrow$ **can be solved by a constraint solver, a SAT solver**

Note:

- $\forall i \in [1, n]$ in the first group of constraint is in fact $\bigwedge_{j \in [1,n]}$
- $j, k \in [1, n], j \neq k$
  can be changed to
  $j \in [1, n-1], k \in [j+1, n]$
  to avoid redundant constraints (commutativity of $\vee$)

n-queen: a third model

- Variables and Domains: for $i \in [1, n]$
  - $c_i \in [1, n]$ (column position of the i-th queen)
  - $\ell_i \in [1, n]$ (line position of the i-th queen)
- Constraints:
  - queens i and j are not on the same line:
    $\forall i, j \in [1, n], i \neq j, \quad \ell_i \neq \ell_j$
  - queens i and j are not on the same column:
    $\forall i, j \in [1, n], i \neq j, \quad c_i \neq c_j$
  - queens i and j are not on the same diagonal:
    $\forall i, j \in [1, n], i \neq j, \quad \ell_i - \ell_j \neq c_i - c_j$ and $\ell_i - \ell_j \neq c_j - c_i$

About the third model

Model:

- two 1-d array (vectors) of coordinates
- a queen is given by 2 variables (coordinate row/column) (c_i, ℓ_i) coordinates of queen k

Family:

- in the Integer Linear Programming family
- or integer Finite Domain (FD) model

$\Rightarrow$ **can be solved by an Integer FD constraint solver**

Note:

- a lot of symmetric solutions
- e.g., values of (c_i, ℓ_i) and (c_j, ℓ_j) can be switched
- globally: queen number k can be put in (c_i, ℓ_i), whatever i

n-queen: a fourth model

- Variables and Domains: $x_i \in [1, n]$ for $i \in [1, n]$
  x_i is the i-th row, x_i's value is the column of the i-th queen
  i.e., $x_i = j$ means: the i-th queen is on row i and column j
- Constraints:
  - one and only one queen per column:
    $\forall i, j \in [1, n], i \neq j, \quad x_i \neq x_j$
  - one and only one queen per row:
    already in the semantics of variables and domains
  - not 2 queens on a diagonal
    $\forall i, j \in [1, n], i \neq j, \quad x_i - x_j \neq i - j$ and $x_i - x_j \neq j - i$

About the fourth model

Model:

- one 1-d array (vector) of columns
- a queen is given by 1 variable
  x_i is the place of queen i

Family:

- same family as the third model, and same remarks
- much less symmetric solutions

Let's improve it

n-queen: a fifth model

- Variables and Domains: $x_i \in [1, n]$ for $i \in [1, n]$
  $x_i = j$ means: the i-th queen is on row i and column j
- Constraints:
  - one and only one queen per column:
    $alldifferent(\{x_1, \ldots, x_n\})$
  - one and only one queen per row:
    already in the semantics of variables and domains
  - not 2 queens on a diagonal
    $alldifferent(\{x_i + i | i \in [1..n]\})$
    $alldifferent(\{x_i - i | i \in [1..n]\})$

About the fifth model

- integer FD family
- use of global constraints (*alldifferent*):
  - simpler modeling, more declarative, more concise
  - but also, better solving (specific algorithm)

n-queen: a 6th model

- Variables: $R_i, C_i, \quad i \in [1, n]$
  Rows, Columns, and Cells of the chessboard
- Domains: $\{q_1, \ldots, q_n\}$
- Constraints:
  - one and only one queen per row:
    $\forall i \in [1, n], \quad R_i = \{q_i\}$
  - one and only one queen per column:
    $\forall i \in [1, n], \quad |C_i| = 1$ and $|\bigcup_{i=1}^{n} C_i| = n$
  - not 2 queens on a diagonal
    $|(R_1 \cap C_2) \cup (R_2 \cap C_1)| \leqslant 1$
    $\ldots$

About the 6th model

Model:

- two 1-d array (vectors) of columns and rows
- a queen is given by the intersection of 2 variables
  queen *i* is at the intersection of row *i* and a column to be determined

Family:

- family of set constraints
- a type of FD variables
- solved by a set constraint CP solvers

And it works!

Model in PyCSP3 (Python)

```
from pycsp3 import *

n = data  # number of queens, it's a data

# q[i] is the column where is put the ith queen (at row i)
q = VarArray(size=n, dom=range(n))

satisfy(
    AllDifferent(q),  # no 2 queens on the same column

    # no two queens on the same upward diagonal
    AllDifferent(q[i] + i for i in range(n)),

    # no two queens on the same downward diagonal
    AllDifferent(q[i] - i for i in range(n)))
```

Solving an instance

Create an xml file containing the instance:

```
python .\Queens.py -data=100
```

Solve the instance:

```
java -jar  .\ACE-21-04.jar .\Queens-100.xml
```

Create and solve the instance:

```
python .\Queens.py -data=100 -solve
```

Model in MiniZinc

```
include "alldifferent.mzn";

int: n; % The number of queens.

array [1..n] of var 1..n: q; % The chessboard

constraint alldifferent(q);  % no 2 queens on a column

% no two queens on the same upward diagonal
constraint alldifferent(i in 1..n)(q[i] + i);

% no two queens on the same downward diagonal
constraint alldifferent(i in 1..n)(q[i] - i);

solve satisfy;
```

To be noticed

- **instance = model + data**
  a model represents a family of problem instances
- PyCSP3 and MiniZinc are modelers
  - modelers or modeling languages
  - i.e., languages to model problems using some form of control and abstraction.

  ⇒ from model + data, they generate the instance
- the generated instance can be solved with various solvers
- solvers without “specific” modelers: Gecode, Choco, Oscar, ECLiPSe, . . .

CSP solving

Constraint solving

- Mathematical programming (LP, ILP, NLP, . . .)
  - generally efficient
  - no global constraints
  - no declarative modeling
- Meta-heuristics
  - efficient when adapted to the problem
  - need problem specific representation and operators
  - not so "generic"
  - incomplete
- Constraint propagation-based solvers
  - generic, i.e., problem independent
  - generic, i.e., same technique for variable types
  - enables declarative modeling
  - more efficient with global constraints

Constraint propagation-based solver

- need to explore the search space
- interleaves two steps:
  - reduction of the search space, i.e., constraint propagation
  - exploration of the search space, i.e., search
- constraint propagation
  - fix-point application of filtering function
  - a filtering function removes values of variable domains that cannot participate in any solution
- search
  - split the current search space
  - decide which "branch" to explore

Sketch of a solver

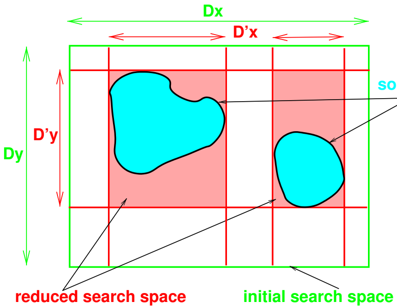
function solve($\mathcal{P} = (X, D, C) : CSP$) $\longrightarrow Map < X, D >$
S: Set(CSP)
$\mathcal{P}'$: CSP
    $S \leftarrow \{\mathcal{P}\}$
    while $S \neq \emptyset$
        $\mathcal{P}' \leftarrow \text{select}(S)$ # select a CSP
        $S \leftarrow S \setminus \{\mathcal{P}'\}$ # remove it from S
        $\mathcal{P}' \leftarrow$ propagate($\mathcal{P}'$) # reduce search space of $\mathcal{P}'$
        if *happy* # expected solution found
            then return solution($\mathcal{P}'$)
            elif $\neg unsat(\mathcal{P}')$ # there may still be solutions in $\mathcal{P}'$
                then $S \leftarrow S \cup split\_search\_space(\mathcal{P}')$
                    # split into “smaller” CSPs, and add them to S
    endWhile
    return $\emptyset$

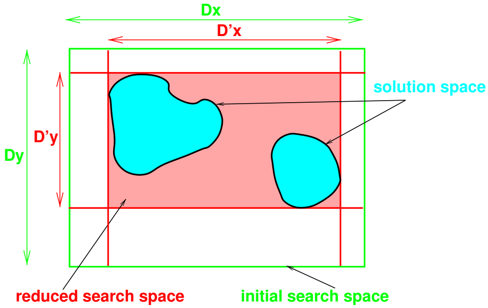
Filtering

Filtering domains of variables

- constraint propagation = fix-point of filtering functions
  the work of a function can wake up other functions
- a constraint can be seen as a sub-problem
- inconsistent values are removed/filtered using a constraint
- several level/strength of filtering
  (resulting property is called local consistency)
  - AC (Arc Consistency): all inconsistent values are deleted
  - BC (Bound Consistency): only inconsistent bounds of domains are deleted
  - ...

AC vs. BC


Dx
D'x
Dy
D'y
solution space
reduced search space
initial search space


Dx
D'x
Dy
D'y
solution space
reduced search space
initial search space

Filtering examples

- Constraint $w + 3 = z$ with
  - $dom(w) = \{0, 1, 3, 4, 5\}$
  - $dom(z) = \{4, 5, 8\}$
- After AC filtering:
  - $dom(w) = \{1, 5\}$
  - $dom(z) = \{4, 8\}$
- After BC filtering:
  - $dom(w) = \{1, 3, 4, 5\}$
  - $dom(z) = \{4, 5, 8\}$

BC (AC) filtering for $x \leqslant y$ (integer FD)

- constraint:

$$x \leqslant y$$

- intuitively:

$$x \leqslant \max D_y$$
$$y \geqslant \min D_x$$

- BC with initial domains $D_x = [a, b], D_y = [c, d]$

$$D_x \leftarrow [a, \min\{b, d\}]$$
$$D_y \leftarrow [\max\{a, c\}, d]$$

BC filtering for $x = y + z$ (Integer FD)

- constraint:

$$x + y = z \quad \equiv \quad x = z - y \quad \equiv \quad y = z - x$$

- intuitively:

$$\begin{aligned} z &\geqslant \min D_x + \min D_y & z &\leqslant \max D_x + \max D_y \\ x &\geqslant \min D_z - \max D_y & x &\leqslant \max D_z - \min D_y \\ y &\geqslant \min D_z - \max D_x & y &\leqslant \max D_z - \min D_x \end{aligned}$$

- BC from $D_x = [a, b], D_y = [c, d], D_z = [e, f]$

$$\begin{aligned} D_x &\leftarrow D_x \cap [e - d, c - f] \\ D_y &\leftarrow D_y \cap [e - b, f - a] \\ D_z &\leftarrow D_z \cap [a + c, b + d] \end{aligned}$$

What about global constraints? Alldifferent?

With "simple" constraints:

- constraints: $x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3$
  domains: $x_1 \in [0, 1], x_2 \in [0, 1], x_3 \in [0, 1]$
- no filtering with $x_1 \neq x_2$ (nor with $x_1 \neq x_3$, or $x_2 \neq x_3$)
  - $x_1 = 0$ and $x_2 = 1$, or $x_1 = 1$ and $x_2 = 0$

With a more global view, stronger filtering:

- constraints: $\mathit{alldifferent}(\{x_1, x_2, x_3\})$
  domains: $x_1 \in [0, 1], x_2 \in [0, 1], x_3 \in [0, 1]$
- 3 variables and 2 values $\Rightarrow$ no solution

and

- constraints: $\mathit{alldifferent}(\{x_1, x_2, x_3\})$
  with $x_1 \in [1, 2], x_2 \in [1, 2, 3, 4], x_3 \in [1, 2]$
- filtering: $x_1 \in [1, 2], x_2 \in [3, 4], x_3 \in [1, 2]$

What about global constraints? Alldifferent?

- various filtering algorithms for the *alldifferent* constraint
  - remember "You can solve Sudoku" ⇒ Hall sets
  - or bi-partite graphs
- main global constraints:
  - sum: $\sum_{i=1}^{r} c_i.x_i \triangleq L$ ($\triangleq \in \{=, \neq, \geqslant, \ldots\}$)
  - cumulative
  - cardinality
  - element
  - ...

Constraint propagation

Propagation

- iterated application of filtering functions
- untill reaching a fixed-point
- or untill a given criterion
  (before reaching local consistency)

Propagation algorithm:

- parameters:
  - $D = D_1 \times D_2 \times \ldots \times D_n$: Cartesian product of domains
  - $F = \{f_1, f_2, \ldots, f_n\}$: a set of filtering functions
- result: D: Cartesian product of reduced domains

Propagation algorithm

function propagate$(D, F) \longrightarrow D$

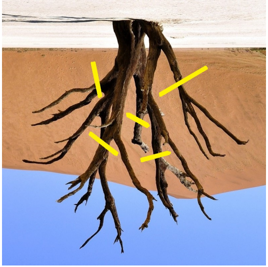
$G \leftarrow \emptyset$
while $F \neq \emptyset$
$F \leftarrow F \setminus \{f_i\}$ # select a function, remove if from F
$G \leftarrow G \cup \{f_i\}$ # add it to F
$D' \leftarrow f_i(D)$ # filter domains with f_i
$F \leftarrow F \cup G'$ and $G \leftarrow G \setminus G'$
where $\forall g \in G, \exists x_k \in var(g), D'_k \neq D_k \rightarrow g \in G'$
functions whose at least one variable
has been modified must be woken
$D \leftarrow D'$
endwhile
return D

Search:
exploration of the search space

Search tree

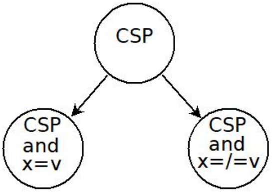
- a complete search can be represented by a “reversed” tree
  - nodes: filtered CSPs
  - root: initial CSP
  - branching corresponds to split (solve algorithm)
  - leaves: success (solution) or fail (unsat) CSP
- goal: pruning the search tree ⇒ to improve solving efficiency
  - cutting branches that do not lead to a success leaf

with pruning:



Pruning and splitting

- pruning by constraint inference: filtering and propagation
- split: split a CSP into sub-CSPs such that no solution is lost
  - generally: labelling
  - a branch with: $x = v, v \in D_x$
  - a second branch with $x \in D_x \setminus \{v\}$
  - selection heuristics: which variable? which value?



Complete exploration

Classical approach

- depth-first left-right
  (find a solution or fail and prune)
- backtracking mechanism
  when a branch fails:
  - go back to last split
  - try the other branch
- interleaving of
  - decisions (branching)
  - inferences (propagation, reduction of the search space)

To go futher on CP

Modelers (and solvers)

- PyCSP3:
  - Modeler: https://github.com/xcsp3team/pycsp3 or https://pypi.org/project/pycsp3/
  - Tutorial: https://github.com/xcsp3team/pycsp3/blob/master/guidePyCSP3.pd
- MiniZinc:
  - Modeler: https://www.minizinc.org/
  - Tutorial: https://www.minizinc.org/doc-2.5.5/en/part_2_tutorial.html

Solvers/Modelers from commercial companies

- Z3 (SMT - Microsoft)
- OR-Tools (Google)
- OPL (IBM)