Introduction to Constraint Solving

Prof. Tias Guns
<tiас.guns@kuleuven.be>
@TiasGuns
Constraint programming

“Solving combinatorial optimisation problems”

- Vehicle Routing
- Scheduling
- Packing
- Other combinatorial problems

[Solved and visualized with the CPMpy constraint solving library]
Constraint solving paradigm

Model + Solve

- Decision variables
- Constraints
- Objective function
Current combinatorial optimisation practice

Model + Solve

Opt. expert → Domain experts

Stakeholders
Research trend

Can we learn it instead?

1) learn to model
2) learn to solve (faster)
1) learn to model

Constraint Acquisition: learn the constraints

Predict-then-optimize/
Decision-focused learning: learn the objective (and more)

Model + Solve

Decision variables
Constraints
Objective function

ML-based stochastic optimisation

Constraining NN output
Neuro-Symbolic AI
2) learn to solve (faster)

Portfolios, automated tuning and algorithm configuration

Learning optimization proxies

Learning to branch, Learning search heuristics

Decision variables
Constraints
Objective function

Model + Solve

Reinforcement Learning during search
Model + Solve examples
Frietkot

Mayonnaise  Ketchup  Curry Ketchup  Andalouse  Samurai
The ‘frietkot’ problem

SAT solving

Input: Boolean formula in ‘Conjunctive Normal Form’ (CNF)
= list of clauses
SAT solvers

Input: Boolean formula in ‘Conjunctive Normal Form’ (CNF)

Solver: propagation, clause learning and search

- propagate: unit clauses (a ∨ false → a=True)
- clause learning after encountering a failure:
  - maintain implication graph of assignments
  - on conflict (a ∧ NOT a), resolve the reason, add as clause
- search: branch on literal (e.g. most active one)
CPMpy demo: PySAT

Frietkot problem

+ “There Are No CNF Problems”
Constraint Programming Research

Rich research on modeling languages, automatic transformations, solver independence, modelling tools

Tools: MiniZinc, Essence’, CPMpy

Rich research on efficient solvers, (global) constraint propagators, automatic search, algorithm configuration, ...

Tools: OrTools, Gecode, Gurobi, ...
Tias’ Belgian beer guide

Stella Artois, from Leuven, 5.2%, must-try factor: 5/10

Duvel, devilish blond, 8.5%, must-try factor: 8/10

Vedett IPA, tastefully hoppy, 6%, must-try factor: 7.5/10

Tripel Karmeliet, strong blond, 8.4%, must-try factor: 8.2/10

Gouden Carolus Whiskey Infused, 11.7%, must-try factor: 9.5/10

Kriek Lindemans, sweet cherry beer, 3.5%, must-try factor: 7/10
Belgian summerschool problem

Which beers to drink, such that you can still pay attention tomorrow?

Model =
- Variables, with a domain
  - st, du, vi, tk, gw, kl :: {0,1}
- Constraints over variables
  - 52*st + 85*du + 60*vi + 84*tk + 117*kl + 35*gw <= 4*52
- Optionally: an objective
  - maximize(50*st + 80*du + 75*vi + 82*tk + 95*kl + 7*gw)

Model.solve()

CPMpy+Pandas demo
# m.solve(): Solving Paradigms

<table>
<thead>
<tr>
<th>Model</th>
<th>SAT</th>
<th>CP</th>
<th>MIP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clauses (no objective)</td>
<td>Clauses, Linear, Global, Linear/any objective</td>
<td>Linear constraints, Linear objective</td>
</tr>
</tbody>
</table>
## Solving Paradigms

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SAT</strong></td>
<td>Clauses (no objective)</td>
<td>Unit Propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clause learning</td>
</tr>
<tr>
<td><strong>CP</strong></td>
<td>Clauses, Linear, Global,</td>
<td>Constraint Propagation</td>
</tr>
<tr>
<td></td>
<td>Linear/any objective</td>
<td>domain eliminations</td>
</tr>
<tr>
<td><strong>MIP</strong></td>
<td>Linear constraints, Linear</td>
<td>Relaxation</td>
</tr>
<tr>
<td></td>
<td>objective</td>
<td>Cutting planes</td>
</tr>
</tbody>
</table>
CP Solving: domain reductions

Constraint Store

- C1: $X \lor Y$
- C2: $A + B \geq 2$
- C3: $B \neq C$
- C4: AllDifferent$(A,B,C)$

Domain Store

Ex: $X = \{0,1\}$, $Y = \{0,1\}$
    $A = \{0,1\}$, $B = \{0,1,2\}$, $C = \{1,2\}$

Circuit()
Cumulative()
CP Solving: domain reductions

**Constraint Store**

- **C1:** $X \lor Y$
- **C2:** $A + B \geq 2$
- **C3:** $B \neq C$
- **C4:** AllDifferent($A,B,C$)

**Domain Store**

Ex: $X = \{0,1\}$, $Y = \{0,1\}$
$A = \{0,1\}$, $B = \{0,1,2\}$, $C=\{1,2\}$

**Functions**

- Circuit()
- Cumulative()
CP Solving: domain reductions

**Constraint Store**

- C1: $X \lor Y$
- C2: $A + B \geq 2$
- C3: $B \neq C$
- C4: AllDifferent($A,B,C$)

**Domain Store**

Ex: $X = \{0,1\}, Y = \{0,1\}$

$A = \{0,1\}, B = \{0,1,2\}, C = \{1,2\}$

- Circuit()
- Cumulative()
CP Solving: domain reductions

**Constraint Store**

C1: \( X \lor Y \)

C2: \( A + B \geq 2 \)

C3: \( B \neq C \)

C4: \( \text{AllDifferent}(A, B, C) \)

... 

**Domain Store**

Ex: \( X = \{0,1\}, \ Y = \{0,1\} \)

\( A = \{0,1\}, \ B = \{1,2\}, \ C = \{1,2\} \)
CP Solving: domain reductions

Constraint Store

C1: \( X \cup Y \)
C2: \( A + B \geq 2 \)
C3: \( B \neq C \)
C4: \( \text{AllDifferent}(A, B, C) \)

Domain Store

Ex: \( X = \{0,1\} \), \( Y = \{0,1\} \)
\( A = \{0,1\} \), \( B = \{0,1,2\} \), \( C = \{1,2\} \)

Till fixpoint

Circuit()
Cumulative()
CP Solving: domain reductions

**Constraint Store**
- **C1:** \( X \lor Y \)
- **C2:** \( A + B \geq 2 \)
- **C3:** \( B \neq C \)
- **C4:** AllDifferent\((A,B,C)\)

**Domain Store**
- **Ex:**
  - \( X = \{0,1\} \)
  - \( Y = \{0,1\} \)
  - \( A = \{0,1\} \)
  - \( B = \{0,1,2\} \)
  - \( C = \{1,2\} \)

**Search**

**Branching**

**Circuit()**

**Cumulative()**

Till fixpoint
Newer: CP-SAT Solving

Constraint Store

C1: X OR Y
C2: A + B >= 2
C3: B != C
C4: AllDifferent(A,B,C)

Domain Store

Ex: X = \{0,1\}, Y = \{0,1\}
   A = \{0,1\}, B = \{0,1,2\}

CNF Store

Incl. domain mapping

Search

CDCL

Circuit()
### Solving Paradigms

<table>
<thead>
<tr>
<th>Model</th>
<th>Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>Clauses (no objective)</td>
</tr>
<tr>
<td>CP</td>
<td>Clauses, Linear, Global, Linear/any objective</td>
</tr>
<tr>
<td>MIP</td>
<td>Linear constraints, Linear objective</td>
</tr>
</tbody>
</table>
MIP solvers

Relax, cut and search

- Relax: ignore integrality, solve Linear Program to obtain lower bound
- Cut: add constraint that avoids (fractional) solution
- Search: split variable \((x \leq a, x > a)\)
Maximize $Z = 8x_1 + 5x_2$
Constraints:
$9x_1 + 5x_2 \leq 45$
$x_1 + x_2 \leq 6$

MIP: relax, cut and search

- **Objective**: $Z = 41.25$
  - $x_1 = 3.75$, $x_2 = 2.25$

- **Feasible integer solution**:
  - $Z = 39$
    - $x_1 = 3$, $x_2 = 3$

- **Infeasible**: $Z = 40.52$
  - $x_1 = 4.44$, $x_2 = 1$
  - $9x_1 + 5x_2 = 46$

- **Feasible integer solution**:
  - $Z = 42$
    - $x_1 = 4$, $x_2 = 2$

- **Infeasible**: $Z = 40$
  - $x_1 = 5$, $x_2 = 0$

- **Optimal integer solution**:
  - $Z = 40$
    - $x_1 = 4$, $x_2 = 1$
## Solving Paradigms

<table>
<thead>
<tr>
<th>Model</th>
<th>Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>Clauses (no objective)</td>
</tr>
<tr>
<td>CP</td>
<td>Clauses, Linear, Global, Linear/any objective</td>
</tr>
<tr>
<td>MIP</td>
<td>Linear constraints, Linear objective</td>
</tr>
</tbody>
</table>
Modeling differences

- CP: high level, problem structure more explicit

- MIP: low level, *relaxable* and as linear constraints
  (some modeling support in commercial solvers)

- SAT: low level, often need to write your own clause generators
How to choose between SAT/MIP/CP solvers?

No free lunch!

General guidelines:

- if decision problem: try SAT first
- if inherently Boolean: try (max)SAT first
- if few constraints or natural to relax: try MIP first
- if suitable globals or complex constraints: try CP first
Modeling: practical considerations

- Model size: MIP/SAT formulations can grow very large (millions of constraints)
- Modeling alternatives:
  - often different ways of modeling same (sub)problem
  - modeling choices matter, needs to be chosen experimentally
- Symmetric solutions and symmetry breaking
Yet another Belgian problem
Perception-based Constraint Solving: a demo application

Tia Gans, Emilia Famba,
Maxime Malambo Re Tondana,
Ignace Bleda, Sorne Beneden,
& Milan Pena

http://sudoku-assistant.cs.kuleuven.be
1) Recognizing the Sudoku digits

- Cut into 81 pieces (introduces additional noise)
- Predict 1-9 or empty (printed and handwritten, robust to borders and markings)
- Custom but standard ML
2) solving the sudoku

Rules of Sudoku (source: sudoku.com)

- Sudoku Rule № 1: Use Numbers 1-9

Sudoku is played on a grid of 9 x 9 spaces. Within the rows and columns are 9 “squares” (made up of 3 x 3 spaces). Each row, column and square (9 spaces each) needs to be filled out with the numbers 1-9, without repeating any numbers within the row, column or square. Does it sound complicated? As you can see from the image below of an actual Sudoku grid, each Sudoku grid comes with a few spaces already filled in; the more spaces filled in, the easier the game – the more difficult Sudoku puzzles have very few spaces that are already filled in.
2) solving the sudoku

Model =

- Variables, with a domain
  - grid[i,j] :: {1..9} for i,j in {1..9}

- Constraints over variables
  - alldifferent(grid[i,:]) for i in {1..9} — rows
  - alldifferent(grid[:,j]) for j in {1..9} — columns
  - alldifferent(square(grid, k,l)) for k,l in {1..3} — squares

  grid[i,j] == given[i,j] if given[i,j] not empty for i,j in {1..9}

Model.solve()
2) solving the sudoku

```python
model = Model()

# Variables
puzzle = intvar(1, 9, shape=given.shape, name="puzzle")

# Constraints on rows and columns
model += [AllDifferent(row) for row in puzzle]
model += [AllDifferent(col) for col in puzzle.T]

# Constraints on blocks
for i in range(0, 9, 3):
    for j in range(0, 9, 3):
        model += AllDifferent(puzzle[i:i+3, j:j+3])

# Constraints on values (cells that are not empty)
model += (puzzle[given!=e] == given[given!=e])

model.solve()
```
Global Constraints: AllDifferent

AllDifferent(), is what?

- “Each variables must have a different value”

- Can be **decomposed** into simpler constraints:

  \[
  \text{AllDifferent}(x_1, x_2, x_3) \iff (x_1 \neq x_2) \land (x_1 \neq x_3) \land (x_2 \neq x_3)
  \]

  For \( n \) variables, \( n^*(n-1)/2 \) pairwise inequalities
Example global constraint: alldifferent

\[ \text{AllDifferent}(x_1, x_2, x_3, x_4) \iff x_1 \neq x_2, x_1 \neq x_3, \ldots, x_3 \neq x_4 \]

Initial domain

Source: A Hybrid AC3-Tabu Search Algorithm for Solving Sudoku Puzzles.
AllDifferent, only puzzles?

Hotel owner: has number of rooms available.
Requests come in, with start/end dates.

=> Do I have enough room to fit all requests?

• often: add to existing allocation
• optimisation: reshuffle to find new allocation?

CPMpy+Pandas+Plotly demo
Example: room scheduling (backup slide)

```python
def model_rooms(df, max_rooms, verbose=True):
    n_requests = len(df)

    # All requests must be assigned to one out of the rooms (same room during entire period).
    requestvars = intvar(0, max_rooms-1, shape=(n_requests,))

    m = Model()

    # Some requests already have a room pre-assigned
    for idx, row in df.iterrows():
        if not pd.isna(row['room']):
            m += (requestvars[idx] == int(row['room']))

    # A room can only serve one request at a time.
    # <=> requests on the same day must be in different rooms
    for day in pd.date_range(min(df['start']), max(df['end'])):
        overlapping = df[(df['start'] <= day) & (day < df['end'])]
        if len(overlapping) > 1:
            m += AllDifferent(requestvars[overlapping.index])

    return (m, requestvars)
```
Extending CP: global constraints

Examples:

- AllDifferent(X,Y,Z)
- Cumulative(...) used in scheduling

**model:** succinctly express a substructure

**solve,** with specialised algorithms:

- optimized data structures = more efficient
- (sometimes) more pruning = more effective
3 short slides on CPMpy's design

Design principle:

Aim to be a thin layer on top of solver API

Central concept: CPMpy expression
Design

CPMpy (user code)

Model
- constraints: expression tree
- objective: expression tree

expressions/

No rewriting!
Like a parser

expressions/

Solver Interface

CPM_ortools
CPM_gurobi
CPM_minizinc
CPM_z3
CPM_pysat
CPM_pySDD

solvers/

Only 1-to-1 mapping of supported expressions

Hardest part

transformations/
Transformations in a nutshell

- model
  - toplevel_list()
  - decompose_in_tree() -> simplify_bool()
  - push_down_negation()
  - simplify_bool()
  - flatten()
  - reify_rewrite()
  - only_numexpr_eq()
  - only_bv_implies()
  - linearize()
  - only_positive_bv()

  - CP lang (MiniZinc)
  - SMT (Z3)
  - BDD (PySDD)
  - SAT (PySAT)
  - CP (OR-Tools)
  - ILP (Gurobi,Exact)
Solvers

CPMpy only interfaces to Python APIs

Key principle: solver can implement any subset of expressions!

Solvers can also choose to:

- Support assumptions or not
- Be incremental or not
- Expose own solver parameters

Currently:
- ortools
- pysat
- minizinc
- gurobi
- pySDD
- Z3
- Exact

Wishlist: GCS, Choco, CPOptimiser, Mistral2, Gecode
More Belgian problems...

- You want to do a guide tour through the city of Leuven, and visit key highlights.

- What is the shortest tour that visits each highlight exactly once, and returns to the starting point?
Travelling Salesman problem

- CP: with a ‘Circuit’ global constraints
  (can also be used for price-collecting TSP, and other variants: just add constraints)

- MIP: ex. MTZ formulation (avoid disconnected components)

CPMpy+Pandas+Geopy+Plotly demo
Job shop scheduling

CPMpy+Pandas+Plotly demo
Beyond Model + Solve
Wider view

Model + Solve
Wider view: integration

- Model
- Model + Solve
- Machine Learning predictions
- Visualisations
- Explainability
- Master-subproblem algorithms
- Interactive solving
- ...

...
Modern Constraint Solving

Model          +          Solve

Machine Learning predictions
Visualisations

Explainability

Master-subproblem algorithms
Interactive solving

...
The changing role of solvers

Holy Grail: user specifies, solver solves [Freuder, 1997]

I think we reached it… MiniZinc, Essence’

“Beyond NP” → constraint solving as an oracle

- Use solver to solve subproblem of larger (imperative) algorithm
- Iteratively build-up and solve a problem until failure
- Integrate neural network predictions (structured output prediction)
- Generate proofs, explanations, or counterfactual examples, …
What would the ideal constraint solving system be?

- Efficient repeated solving  
  => Incremental

- Use CP/SAT/MIP or any combination  
  => solver independent and multi-solver

- Easy integration with Machine Learning libraries  
  => Python and numpy arrays
What would the ideal constraint solving system be?

- **Efficient repeated solving**
  => Incremental

- **Use CP/SAT/MIP or any combination**
  => solver independent and multi-solver

- **Easy integration with Machine Learning libraries**
  => Python and numpy arrays
Incremental room assignment problem

Assume requests come in sequentially. Compute solution on every new request.

def model rooms(df, max rooms, verbose=True):
    n_requests = len(df)

    # All requests must be assigned to one out of the rooms (same room during entire period).
    requestvars = intvar(0, max_rooms-1, shape=n_requests)

    # Model()

    # Some requests already have a room pre-assigned
    for idx, row in df.iterrows():
        if not pd.isna(row['room']):
            n == (requestvars[idx] == int(row['room']))

    # A room can only serve one request at a time.
    # => requests on the same day must be in different rooms
    for day in pd.date_range(min(df['start']), max(df['end'])):
        overlapping = df[(df['start'] == day) & (df['end'] == day)]
        if len(overlapping) > 1:
            n == AllDifferent(requestvars[overlapping.index])

    return (n, requestvars)
Incrementality

Solving:

- **MIP**: can add constraints, change objective
  (mechanisms not documented, e.g. start from previous basis)

- **SAT**: assumption variables: can be toggled on/off when calling solve
  (reuses learned clauses, variable activity)

- **CP**: if CP-SAT, assumption variables like SAT

- **SMT**: pop/push of constraints (Z3)

Modeling?

- Only if using solver API directly...

- With CPMpy: part of the high-level modeling language!
Multiple solutions

New built-in: `m.solveAll()`

Or MiniSearch-style:

```python
x = intvar(0, 3, shape=2)
m = Model(x[0] > x[1])

while m.solve():
    print(x.value())
    m += ~all(x == x.value())  # block solution
```

Returns True (sol. found) or False (no solution)

Adds constraint to model (even if already solved before)
Non-dominated solutions (disjunctive method)

```python
def disjunctive_method(model, objectives_list):
    while model.solve():
        yield [objective.value() for objective in objectives_list]

        # one of the objectives must be better (assume all minimize)
        model += cpmpy.any([[obj < obj.value() for obj in objectives_list]])
```

CPMpy Land Conservation demo
What would the ideal CP system be?

- Efficient repeated solving
  => Incremental

- **Use CP/SAT/MIP or any combination**
  => solver independent and multi-solver

- Easy integration with Machine Learning libraries
  => Python and numpy arrays
Multi-solver

Same syntax, plus can reuse variables and their values

```python
m_ort = SolverLookup.get("ortools", model_knapsack)
m_ort.solve()
print("\nOrtools: ", m_ort.status(), ": ", m_ort.objective_value(), items.value())

m_grb = SolverLookup.get("gurobi", model_knapsack)
m_grb.solve()
print("\nGurobi: ", m_grb.status(), ": ", m_grb.objective_value(), items.value())

# use ortools to verify the gurobi solution
m_ort += (items == items.value())
print("\nGurobi's is a valid solution according to ortools: ", m_ort.solve())
```

Ortools: ExitStatus.OPTIMAL (0.001146096 seconds) : 32.0 [ True False False True True True True True True]

Gurobi: ExitStatus.OPTIMAL (0.0003108978271484375 seconds) : 32.0 [ True False False True False True True True True]

Gurobi's is a valid solution according to ortools: True
Implicit Hitting Set algorithm (explanation-related)

```python
def OCUS_assum(soft, soft_weights, hard=[], solver='ortools', verbose=1):
    # init with hard constraints
    assum_model = Model(hard)
    # make assumption indicators, add reified constraints
    ind = BoolVar(shape=len(soft), name="ind")
    for i, bv in enumerate(ind):
        assum_model += [bv.implies(soft[i])]
    # to map indicator variable back to soft_constraints
    indmap = dict((v, i) for (i, v) in enumerate(ind))

    assum_solver = SolverLookup.lookup(solver)(assum_model)
    if assum_solver.solve(assumptions=ind):
        return []

    ##
    hs_model = Model{
        # Objective: min sum(x_i * w_i)
        minimize=sum(x_i * w_i for x_i, w_i in zip(ind, soft_weights))
    }

    # instantiate hitting set solver
    hittingset_solver = SolverLookup.lookup(solver)(hs_model)

    while(True):
        hittingset_solver.solve()
        # Get hitting set
        hs = ind[ind.value() == 1]

        if not assum_solver.solve(assumptions=hs):
            return soft[ind.value() == 1]

        # compute complement of model in formula F
        C = ind[ind.value() != 1]
        # Add complement as a new set to hit: sum x[j] * hij >= 1
        hittingset_solver += (sum(C) >= 1)
```

repeatedly compute hitting sets (MIP)

CP/SAT as an oracle

Extract Correction Subset
Conversational Human-Aware Technology for Optimisation

What would the ideal CP system be?

- Efficient repeated solving
  => Incremental

- Use CP/SAT/MIP or any combination
  => solver independent and multi-solver

- Easy integration with Machine Learning libraries
  => Python and numpy arrays
Modern Constraint Solving: an example

Sudoku Assistant

Tias Guns, Milan Pesa, Maxime Mulamba, Ignace Bleukx, Emilio Gamba, Senne Berden
Perception-based constraint solving

Pedagogical instantiation: visual sudoku (naïve)

<table>
<thead>
<tr>
<th></th>
<th>img</th>
<th>accuracy cell</th>
<th>grid</th>
<th>failure rate grid</th>
<th>time average (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>94.75%</td>
<td>15.51%</td>
<td>14.67%</td>
<td>84.43%</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Perception-based constraint solving

What is going on?

- Each cell predicts the maximum likelihood value:
  \[ \hat{y}_{ij} = \arg \max P(y_{ij} = k | X_{ij}) \]

- But you need all 81 predictions (one for each given cell), it is a multi-output problem: together this is the ‘maximum likelihood’ interpretation

- If \( \text{sudoku}(\hat{y}) = False \): no solution, interpretation is wrong...
Perception-based constraint solving

What about the *next* most likely interpretation?
Treat prediction as *joint inference* problem:

\[ \hat{y} = \arg \max \prod_{i,j} P(y_{i,j} = k | X_{i,j}) \quad \text{s.t.} \quad \text{sudoku}(\hat{y}) \]

This is the *constrained* ‘maximum likelihood’ interpretation

=> Structured output prediction

Used e.g. in NLP: [Punyakanok, COLING04]
Perception-based constraint solving

Can we use a constraint solver for that?

\[ \hat{y} = \arg \max_{\prod_{ij} P(y_{ij} = k|X_{ij})} \text{ s.t. } \text{sudoku}(\hat{y}) \]

- Log-likelihood trick:

\[
\min \sum_{(i,j) \in \text{given \{1,\ldots,9\}}} \sum_{k \in \{1,\ldots,9\}} -\log(P_{0}(y_{ij} = k|X_{ij})) \times I[s_{ij} = k] \text{ s.t. } \text{sudoku}(\hat{y}) \]
Perception-based constraint solving

Hybrid: CP solver does joint inference over raw probabilities

<table>
<thead>
<tr>
<th></th>
<th>img</th>
<th>accuracy</th>
<th>failure rate</th>
<th>time average (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>94.75%</td>
<td>15.51%</td>
<td>14.67%</td>
<td>84.43%</td>
</tr>
<tr>
<td>hybrid1</td>
<td>99.69%</td>
<td>99.38%</td>
<td>92.33%</td>
<td>0%</td>
</tr>
<tr>
<td>hybrid2</td>
<td>99.72%</td>
<td>99.44%</td>
<td>92.93%</td>
<td>0%</td>
</tr>
</tbody>
</table>

[Maxime Mulamba, Jayanta Mandi, Rocsildes Canoy, Tias Guns, CPAIOR2020]
Sudoku Assistant demo, continued
Implementation: integration

Frontend:
- React-native
- Only displays results

Backend:
- FastAPI (Python)
- NN Service (PyTorch)
- Solver Service (CPMpy)
- Preloading, caching...
Show solution?

Trivial for CP system (subsecond),
Boring and demotivating for user?

In general: human-aware AI & AI assistants:

- **Support** users in decision making
- Respect human *agency*
- Provide *explanations* and learning opportunities
Constraint solving is more than mathematical abstractions...
Bigger picture
Bigger picture

- Learning implicit user preferences
- Learning from the environment
Bigger picture

- Learning implicit user preferences
- Learning from the environment
- Explaining constraint solving
Bigger picture

- Learning implicit user preferences
- Learning from the environment
- Explaining constraint solving
- Stateful interaction
CHAT-Opt:
Conversational Human-Aware Technology for Optimisation

Towards co-creation of constraint optimisation solutions

- Solver that learns from user and environment
- Towards conversational: explanations and stateful interaction
Sudoku Assistant, explanation steps

Click a cell to see its predicted probabilities.

Click to highlight values:

Show hint
Show solution
Scan Another Sudoku

Hide hint
Show solution
Scan Another Sudoku
Modern Constraint Solving

Machine Learning predictions

Visualisations

Explainability

Model + Solve

Master-subproblem algorithms

Interactive solving
CPMpy transformations in a nutshell

- **model**
  - toplevel_list()
  - decompose_in_tree()
  - push_down_negation()
  - simplify_bool()
  - flatten()
  - reify_rewrite()
  - only_numexpr_eq()
  - linearize()
  - only_positive_bv()

- **CP lang (MiniZinc)**
- **SMT (Z3)**
- **BDD (PySDD)**
- **SAT (PySAT)**
- **CP (OR-Tools)**
- **ILP (Gurobi, Exact)**
# Solving Paradigms

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>Solve</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>Clauses (no objective)</td>
<td>Unit Propagation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clause learning</td>
</tr>
<tr>
<td>CP</td>
<td>Clauses, Linear, Global, Linear/any objective</td>
<td>Constraint Propagation=</td>
</tr>
<tr>
<td></td>
<td></td>
<td>domain eliminations</td>
</tr>
<tr>
<td>MIP</td>
<td>Linear constraints, Linear objective</td>
<td>Relaxation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cutting planes</td>
</tr>
</tbody>
</table>
The end / to be continued

Can we learn it instead?

1) learn to model
2) learn to solve (faster)

Model + Solve

Decision variables
Constraints
Objective function
Enjoy!

Machine Learning For Constraint Programming
ACP SUMMER SCHOOL 2023
Liége, Belgium | July 10 To 14, 2023

https://school.a4cp.org/summer2023/