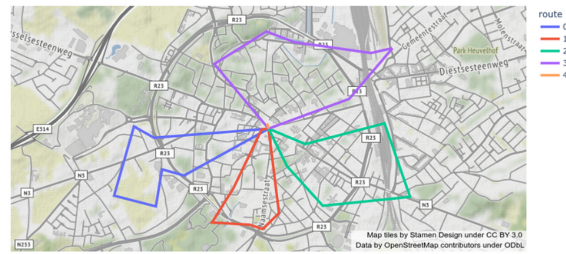
Introduction to Constraint Solving

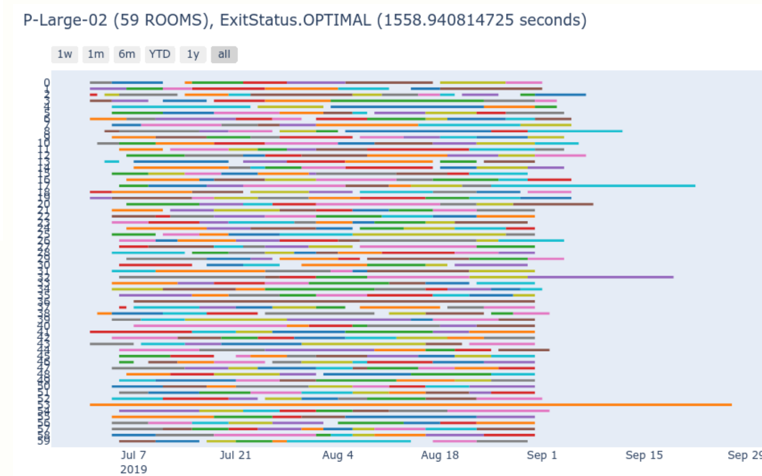
Prof. Tias Guns
<tias.guns@kuleuven.be>
@TiasGuns

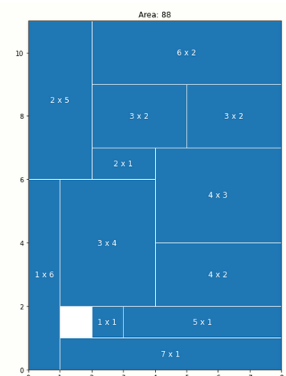
Constraint programming

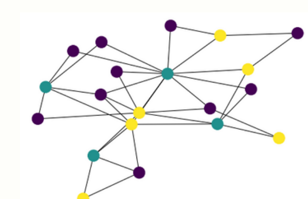
“Solving combinatorial optimisation problems”

- Vehicle Routing
- Scheduling
- Packing
- Other combinatorial problems







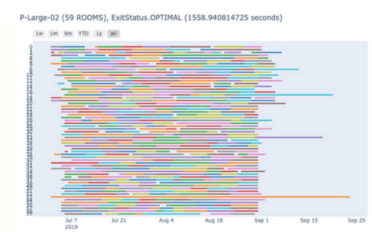


[Solved and visualized with the CPMpy constraint solving library]

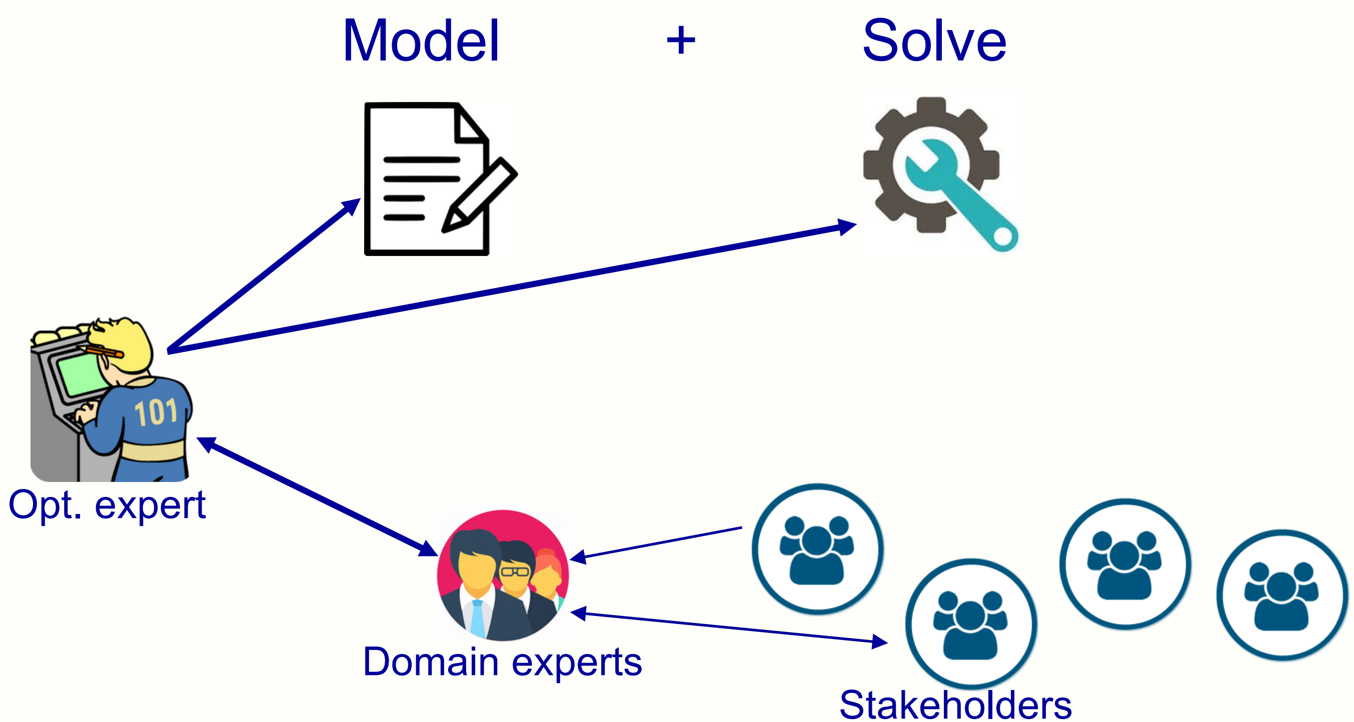
Constraint solving paradigm

Model + Solve

Decision variables
Constraints
Objective function



Current combinatorial optimisation practice



Research trend

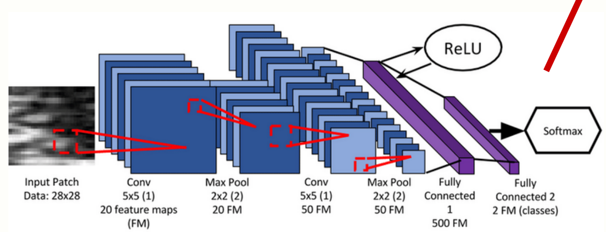
Model + Solve

Decision variables
Constraints
Objective function

1) learn to model

2) learn to solve (faster)

Can we *learn* it instead?



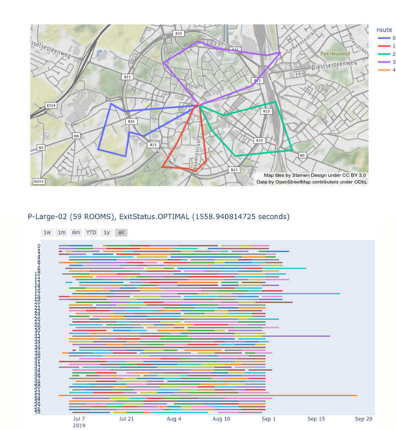
1) learn to model

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

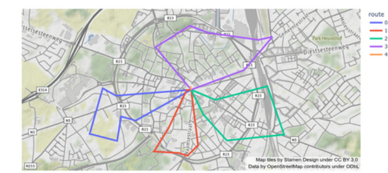
2) learn to solve (faster)

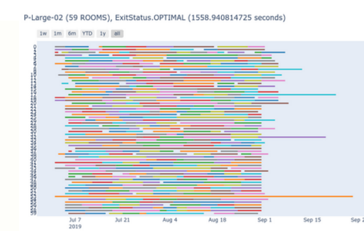
Portfolios, automated tuning and algorithm configuration

Learning optimization proxies

Model + Solve

Decision variables
Constraints
Objective function





Learning to branch,
Learning search heuristics

Reinforcement Learning during search

Model + Solve examples

Frietkot

Mayonnaise

Ketchup

Curry Ketchup

Andalouse

Samurai

The ‘frietkot’ problem

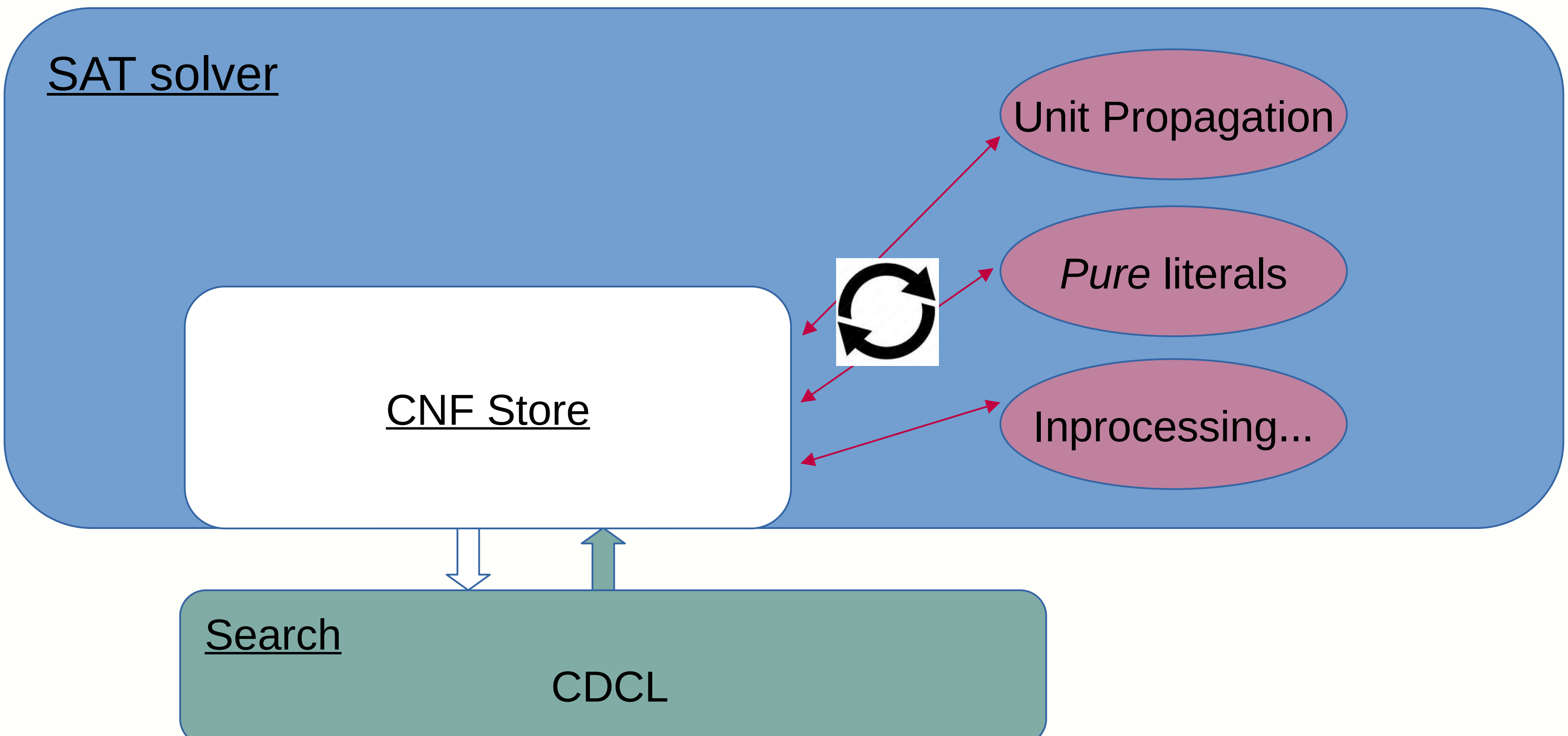
Mayonnaise | Ketchup | Curry Ketchup | Andalouse | Samurai

https://people.cs.kuleuven.be/~tias.guns/frietkot/

SAT solving

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

= list of clauses



SAT solvers

<u>Input:</u> Boolean formula in ‘Conjunctive Normal Form’ (CNF)

<u>Solver: propagation, clause learning and search</u>

- propagate: <u>unit</u> clauses (a $\vee$ false $\rightarrow$ a=True)
- <u>clause learning</u> after encountering a failure:
  - maintain implication graph of assignments
  - on conflict (a $\wedge$ NOT a), resolve the reason, add as clause
- search: <u>branch</u> on literal (e.g. most *active* one)

CPMpy demo: PySAT

Frietkot problem

+ “There Are No CNF Problems”

Constraint Programming Research

Model + Solve

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
- Constraints over variables
- Optionally: an objective

- st, du, vi, tk, gw, kl :: {0,1}
- 52*st + 85*du + 60*vi + 84*tk + 117*kl + 35*gw <= 4*52
- maximize(50*st + 80*du + 75*vi + 82*tk + 95*kl + 7*gw)

Model.solve()

[CPMpy+Pandas demo]

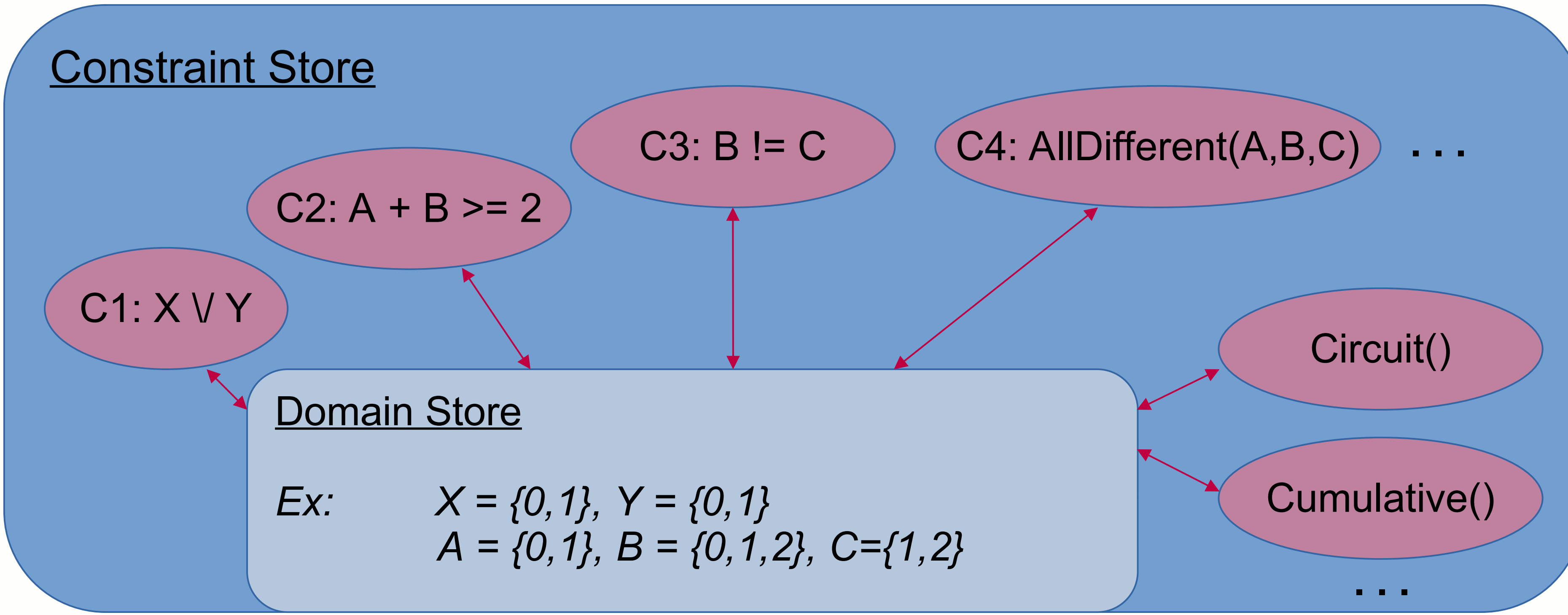
m.solve(): Solving Paradigms

	Model
SAT	Clauses (no objective)
CP	Clauses, Linear, Global, Linear/any objective
MIP	Linear constraints, Linear objective

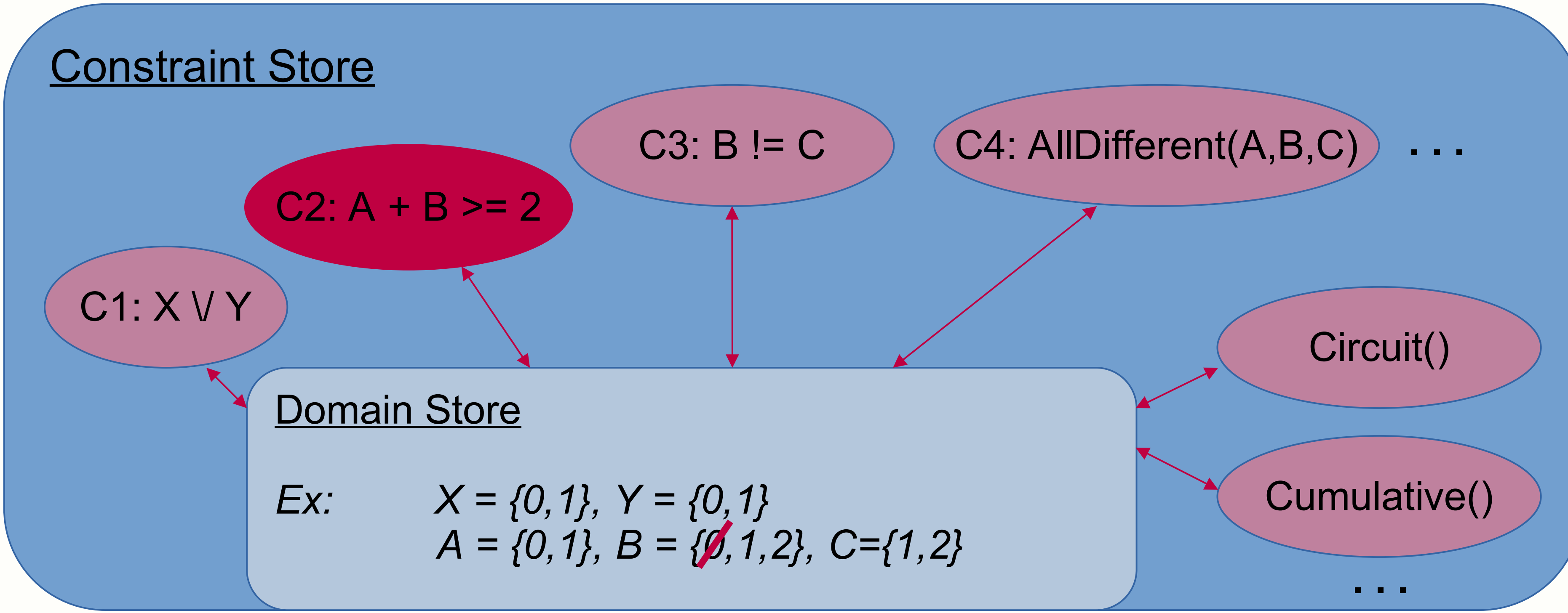
Solving Paradigms

	Model 	Solve 
SAT	Clauses (no objective)	Unit Propagation Clause learning
CP	Clauses, Linear, Global, Linear/any objective	Constraint Propagation= domain eliminations
MIP	Linear constraints, Linear objective	Relaxation Cutting planes

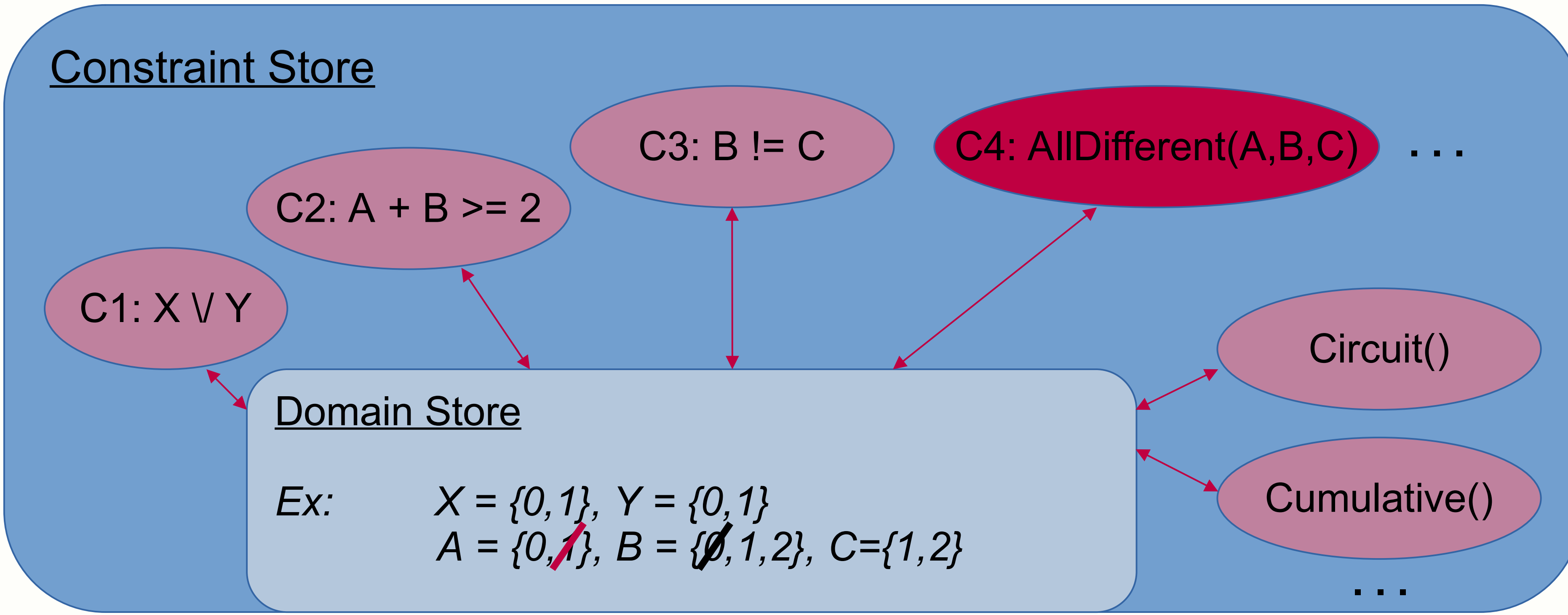
CP Solving: domain reductions



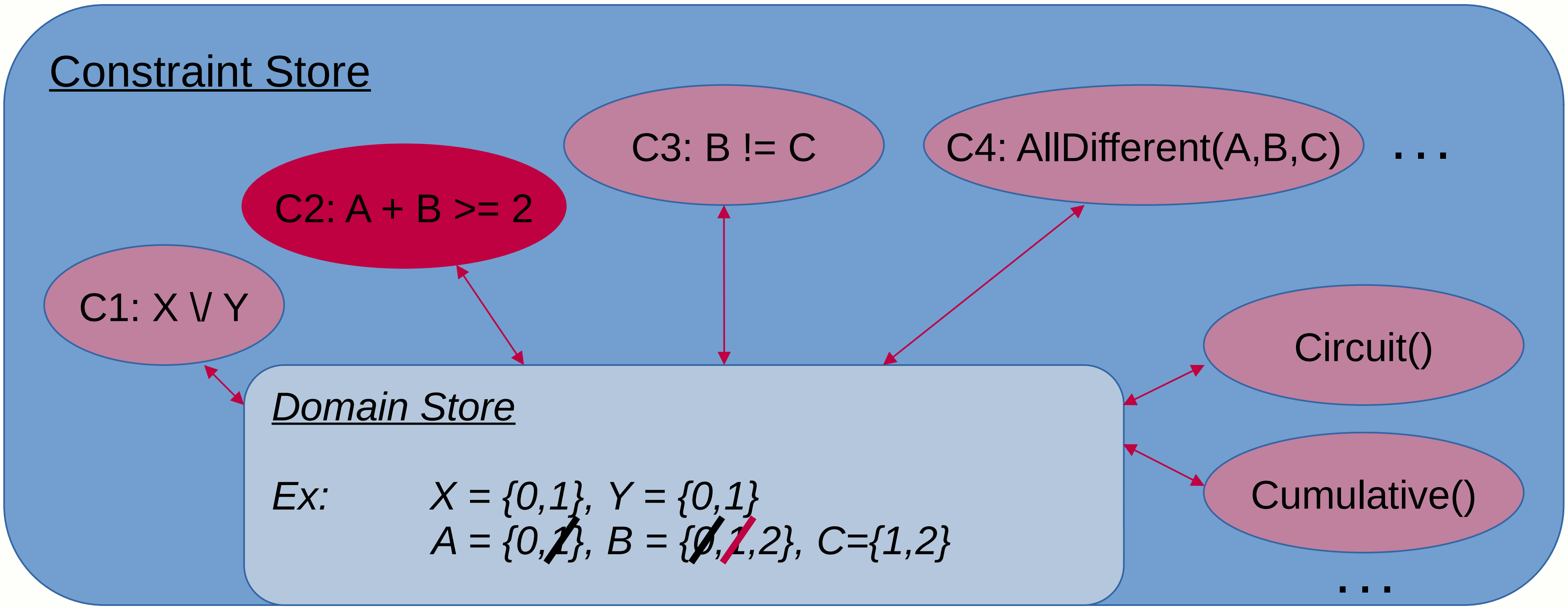
CP Solving: domain reductions



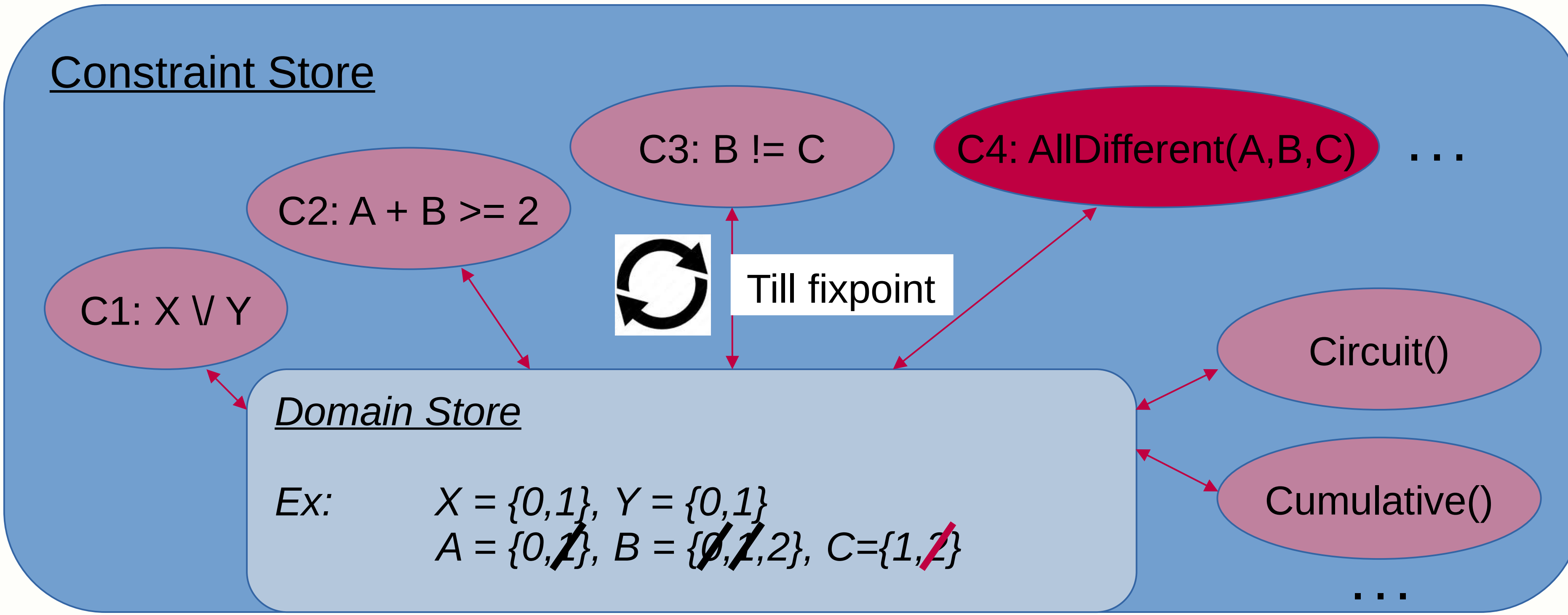
CP Solving: domain reductions



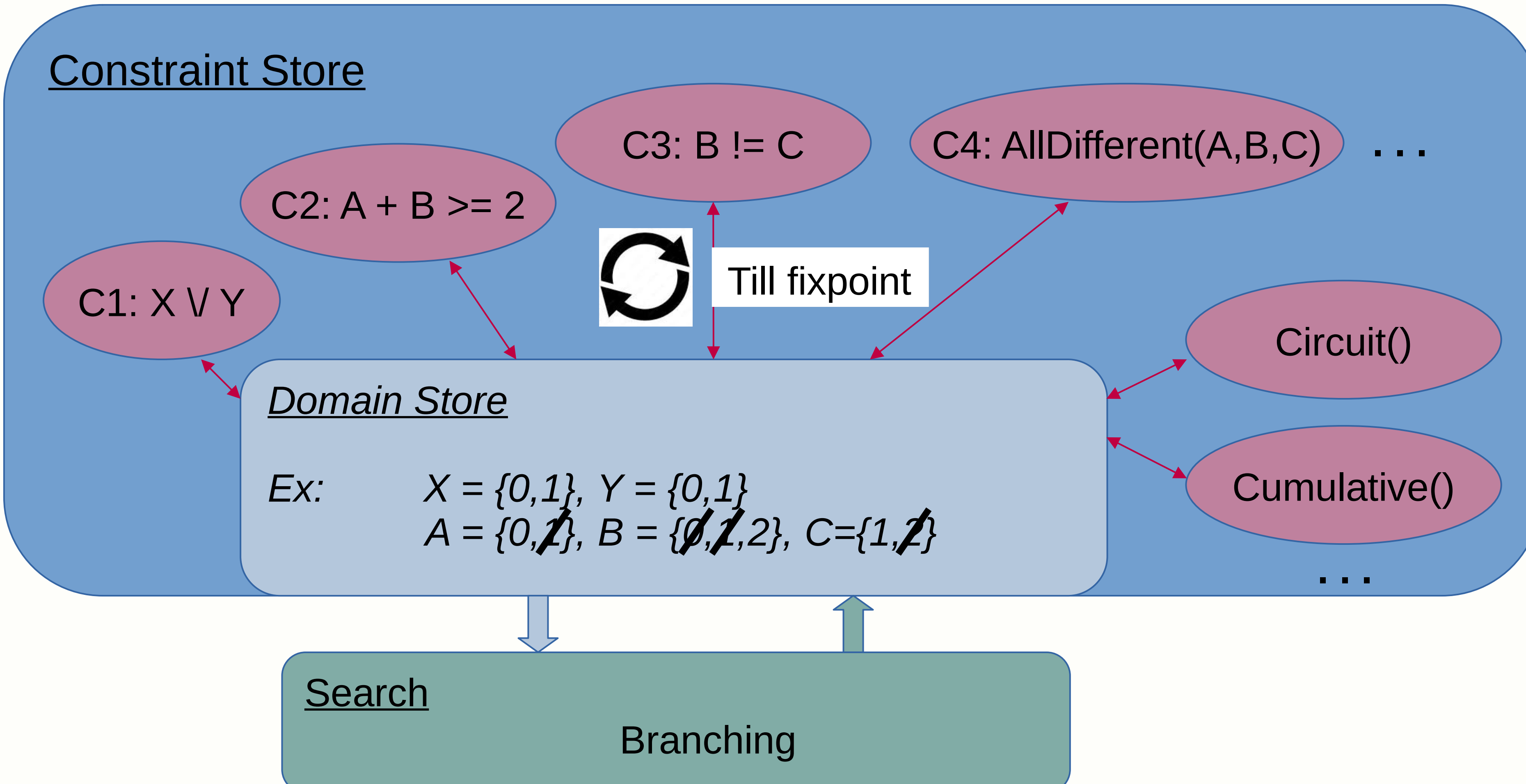
CP Solving: domain reductions



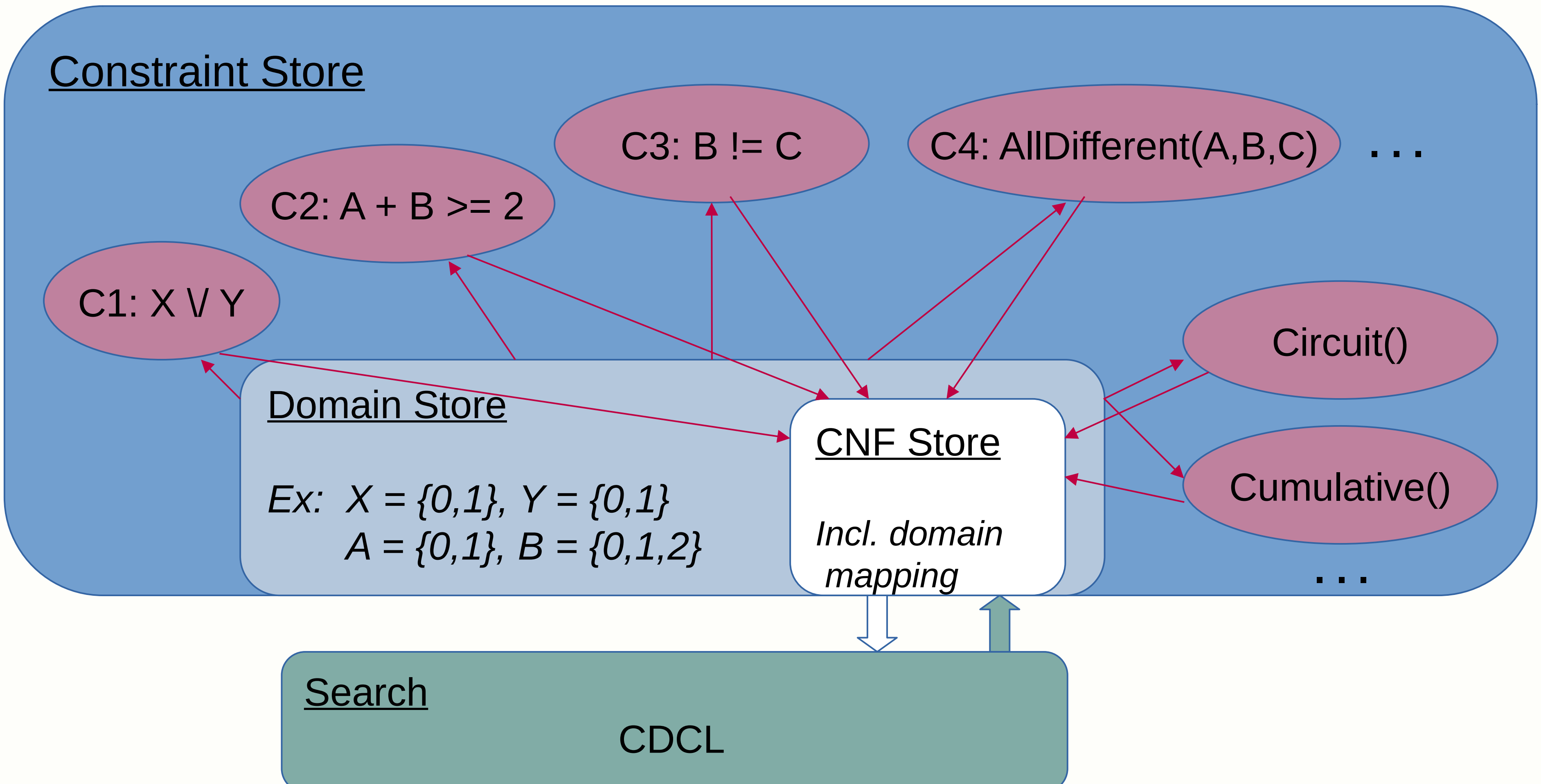
CP Solving: domain reductions



CP Solving: domain reductions



Newer: CP-SAT Solving



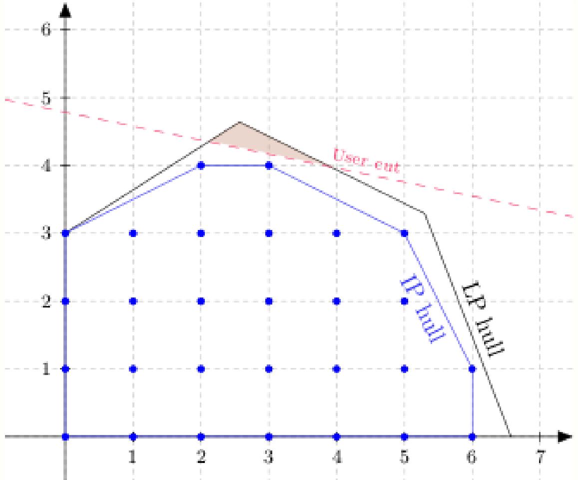
Solving Paradigms

	Model	Solve
SAT	Clauses (no objective)	Unit Propagation Clause learning
CP	Clauses, Linear, Global, Linear/any objective	Constraint Propagation= domain eliminations
MIP	Linear constraints, Linear objective	Relaxation Cutting planes

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 <= a, x > a)



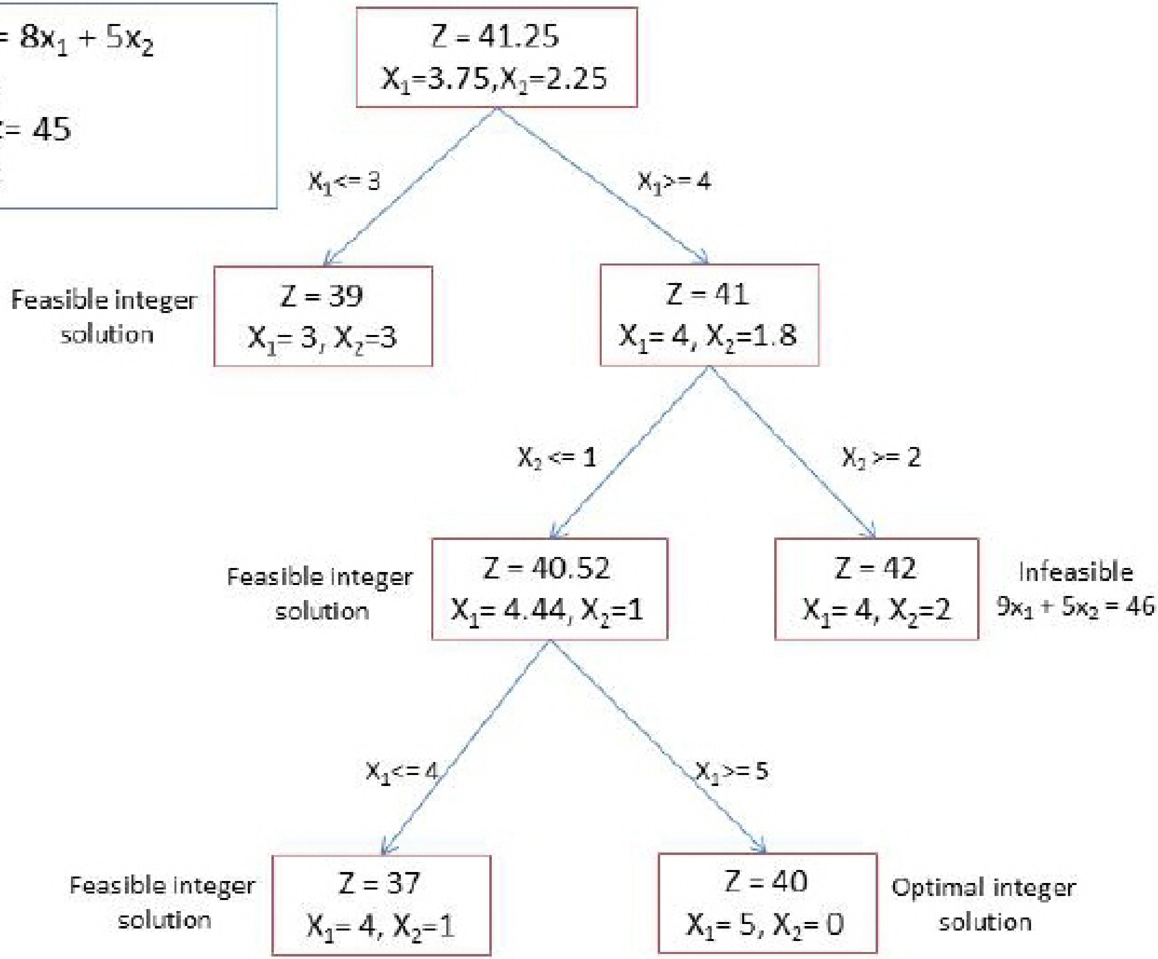
MIP: relax, cut and search

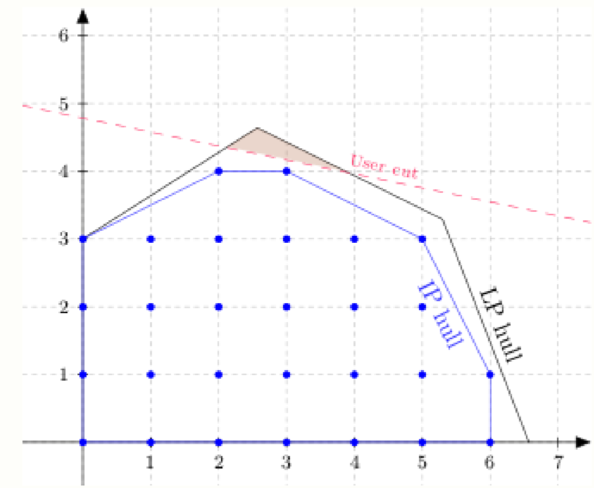
Maximize $Z = 8x_1 + 5x_2$

Constraints:

$9x_1 + 5x_2 <= 45$

$x_1 + x_2 <=6$





Solving Paradigms

	Model	Solve
SAT	Clauses (no objective)	Unit Propagation Clause learning
CP	Clauses, Linear, Global, Linear/any objective	Constraint Propagation= domain eliminations
MIP	Linear constraints, Linear objective	Relaxation Cutting planes

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!

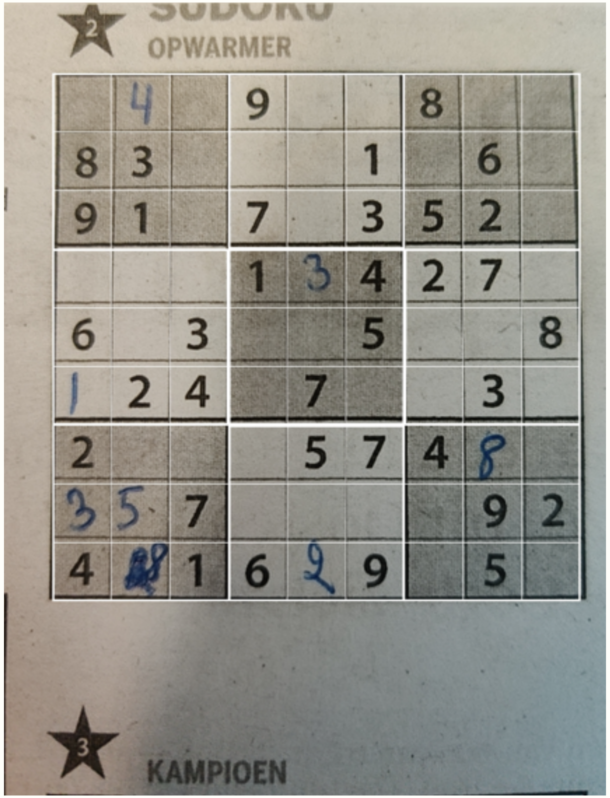
General guidelines:

- 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

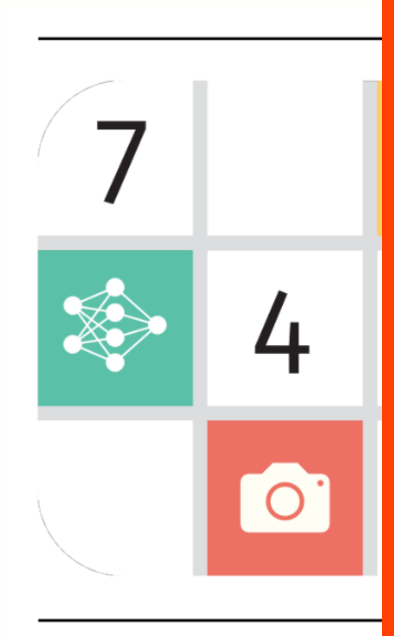
Modeling: practical considerations

- 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



Percep… Solving:



BEST TECHNICAL DEMONSTRATION AWARD

FEBRUARY 7-14, 2023

THE ASSOCIATION FOR THE ADVANCEMENT OF ARTIFICIAL INTELLIGENCE

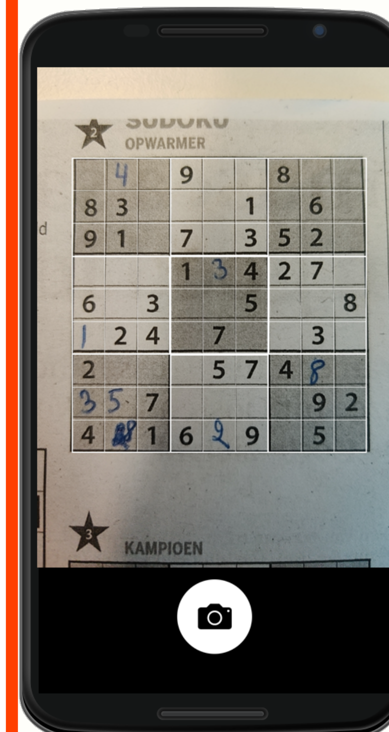
proudly presents

THE AWARD FOR 2023 AAAI BEST TECHNICAL DEMONSTRATION TO

Tias Guns, Emilio Gamba, Maxime Mulamba Ke Tchomba, Ignace Bleukx, Senne Berden, & Milan Pesa

A DEMONSTRATION OF SUDOKU ASSISTANT —
AN AI-POWERED APP TO HELP SOLVE PEN-AND-PAPER SUDOKUS

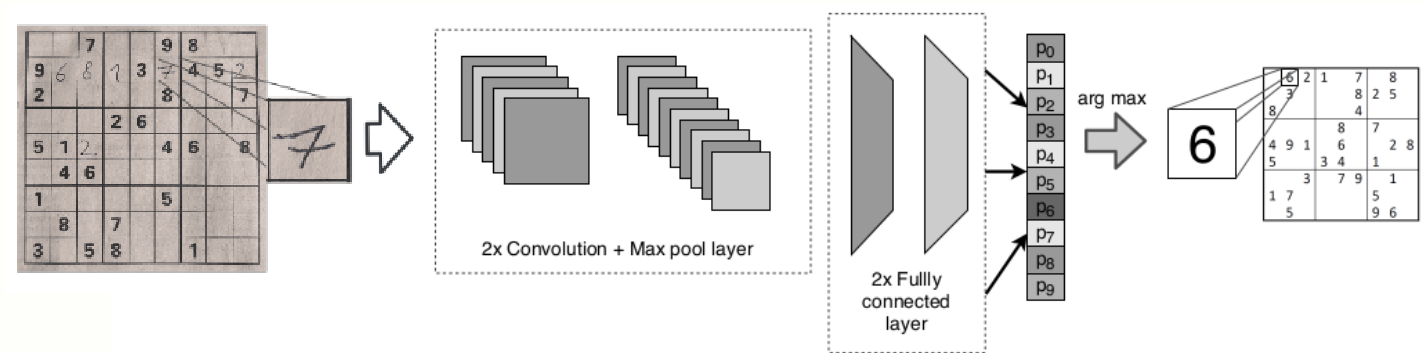
PRESENTED AT THE 37TH AAAI CONFERENCE ON ARTIFICIAL INTELLIGENCE



GET IT ON Google Play

https://s

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.

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

Model + Solve

Decision variables
Constraints
Objective function

Model

2) solving the sudoku

Decision variables
Constraints
Objective function

Model =

- Variables, with a domain
- Constraints over variables

```
- grid[i,j] :: {1..9}          for i,j in {1..9}

- 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()

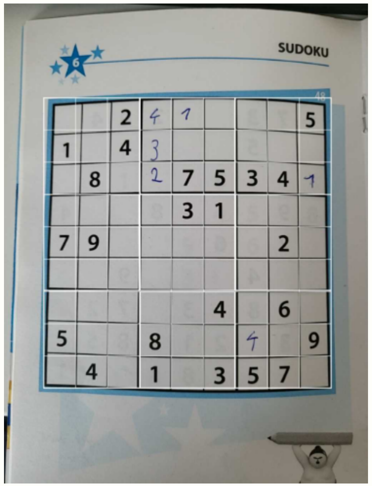
- **Sudoku Rule № 1: Use Numbers 1-9**

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

Model

2) solving the sudoku

Decision variables
Constraints
Objective function




```
e = 0 # value for empty cells
given = np.array([
    [e, e, 2,  4, 1, e,  e, e, 5],
    [1, e, 4,  3, e, e,  e, e, e],
    [e, 8, e,  2, 7, 5,  3, 4, 1],

    [e, e, e,  e, 3, 1,  e, e, e],
    [7, 9, e,  e, e, e,  e, 2, e],
    [e, e, e,  e, e, e,  e, e, e],

    [e, e, e,  e, e, 4,  e, 6, e],
    [5, e, e,  8, e, e,  4, e, 9],
    [e, 4, e,  1, e, 3,  5, 7, e]])
```

```
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?

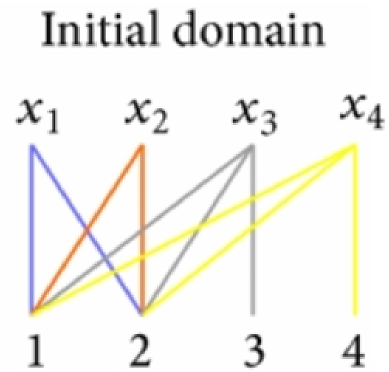
- "Each variables must have a different value"
- Can be **decomposed** into simpler constraints:

$$\text{AllDifferent}(x1,x2,x3) \Leftrightarrow \begin{aligned}&(x_1 \mathrel{!=} x_2)\ \&\\ &(x_1 \mathrel{!=} x_3)\ \&\\ &(x_2 \mathrel{!=} x_3)\end{aligned}$$

For *n* variables, *n*(n-1)/2* pairwise inequalities

Example global constraint: alldifferent

AllDifferent(x1,x2,x3,x4) <=> x_1 != x_2, x_1 != x_3, ..., x_3 != x_4



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.

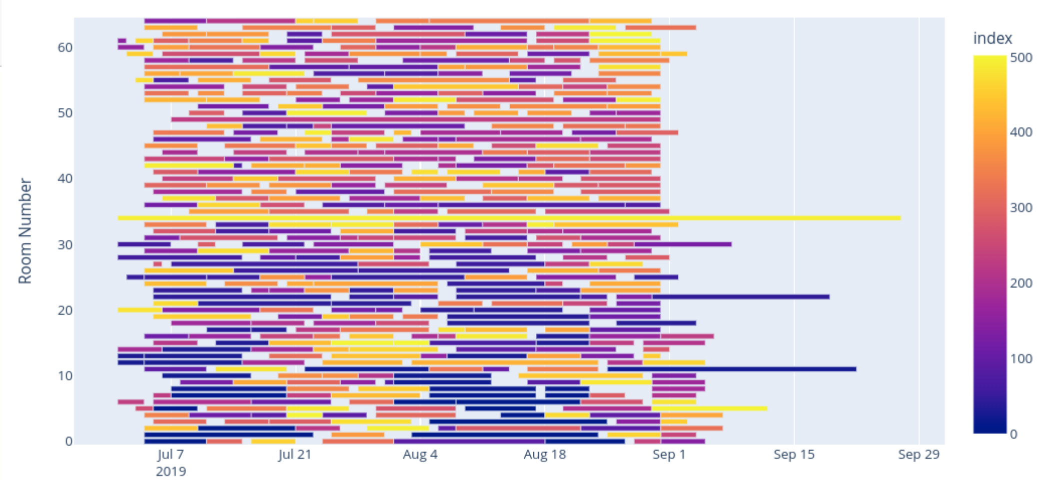
=> Do I have enough room to fit all requests?

- 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)
- *A*[X] = Y   with X,Y variables, A an array “Element”
- Cumulative(...)   used in scheduling

<u>model</u>: succinctly express a substructure

<u>solve</u>, with specialised algorithms:

- optimized data structures = more efficient
- (sometimes) more pruning = more effective

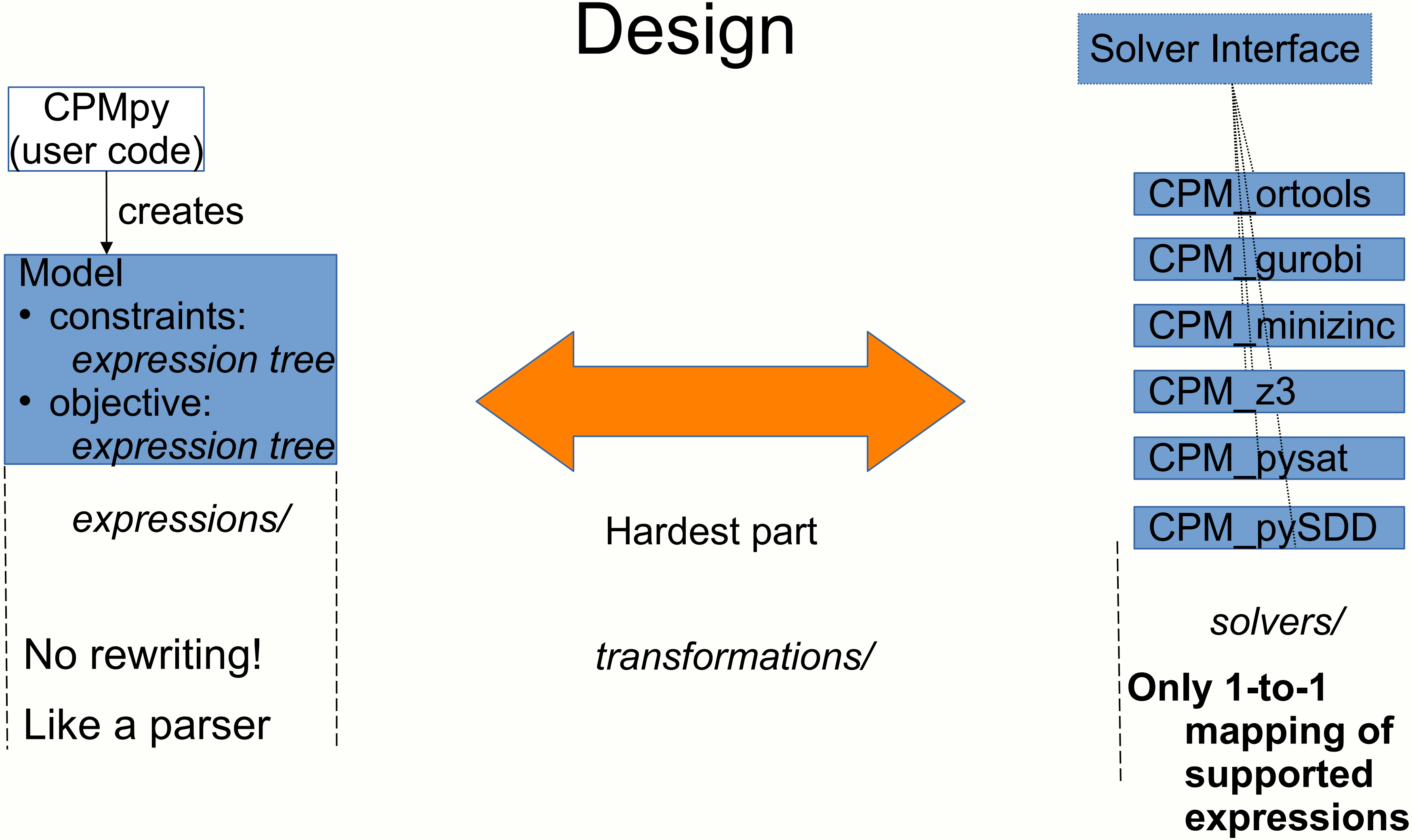
3 short slides on CPMpy's design

Design principle:

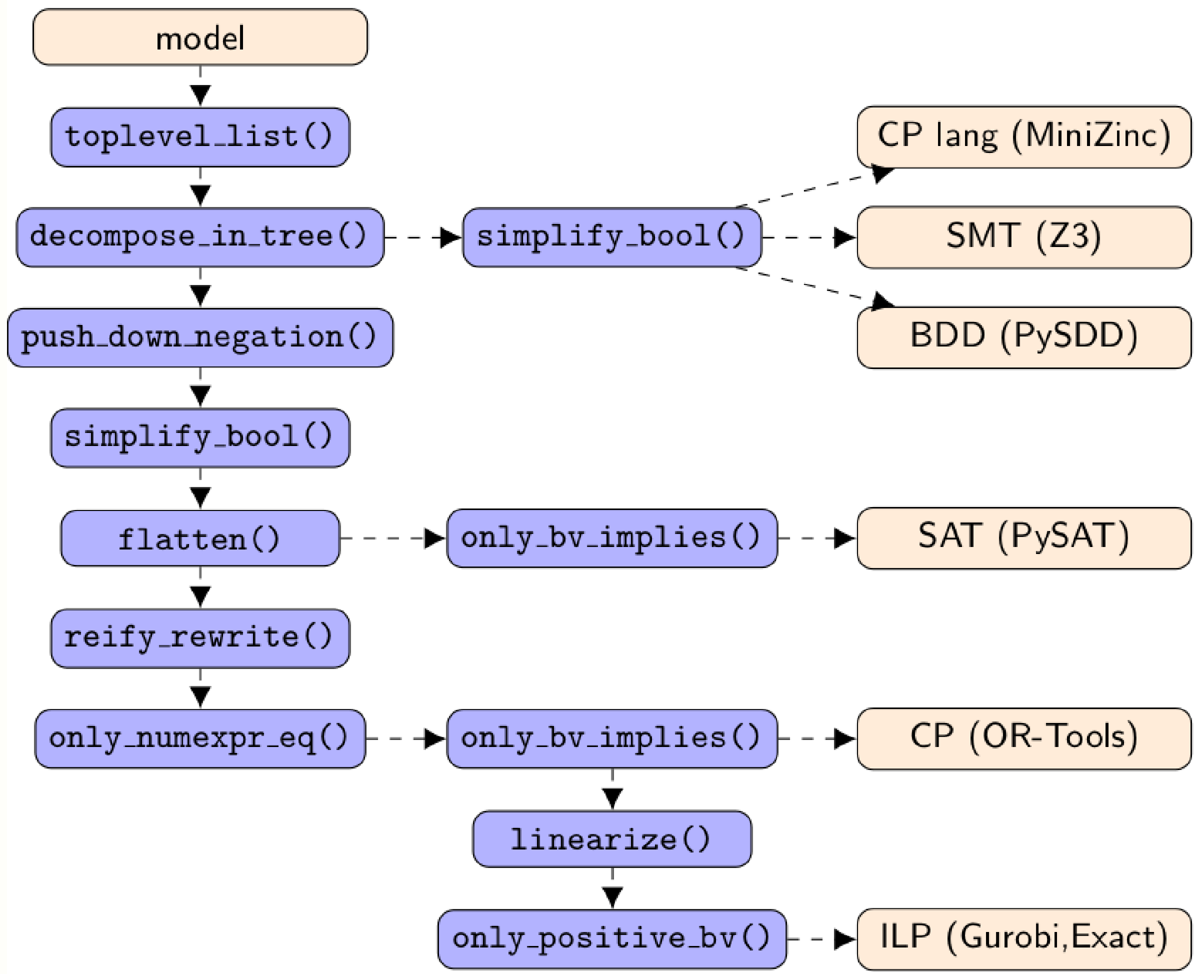
Aim to be a thin layer on top of solver API

Central concept: CPMpy expression

Design


CPMpy
(user code)
creates
Model
• constraints:
expression tree
• objective:
expression tree
expressions/
No rewriting!
Like a parser
Hardest part
transformations/
Solver Interface
CPM_ortools
CPM_gurobi
CPM_minizinc
CPM_z3
CPM_pysat
CPM_pySDD
solvers/
Only 1-to-1
mapping of
supported
expressions

Transformations in a nutshell



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

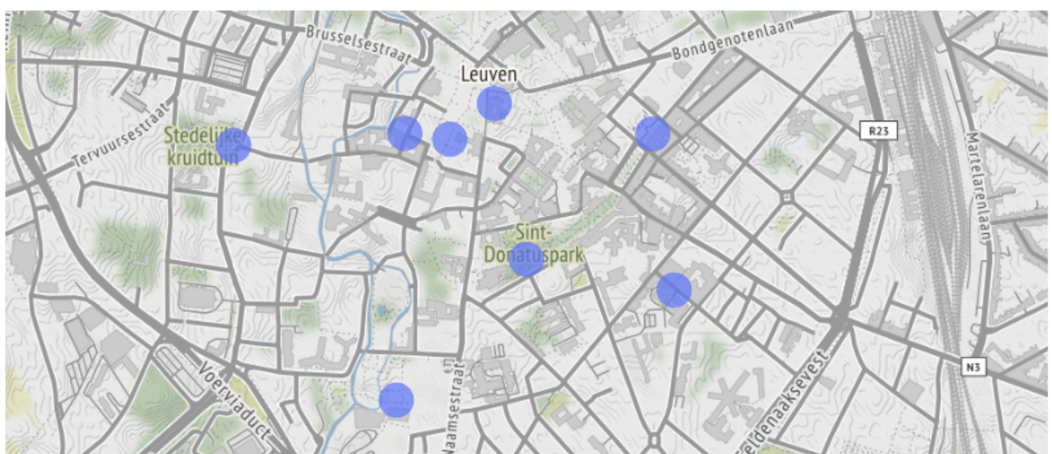
Currently:
- ortools
- pysat
- minizinc
- gurobi
- pySDD
- Z3
- Exact

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?



Traverling 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)

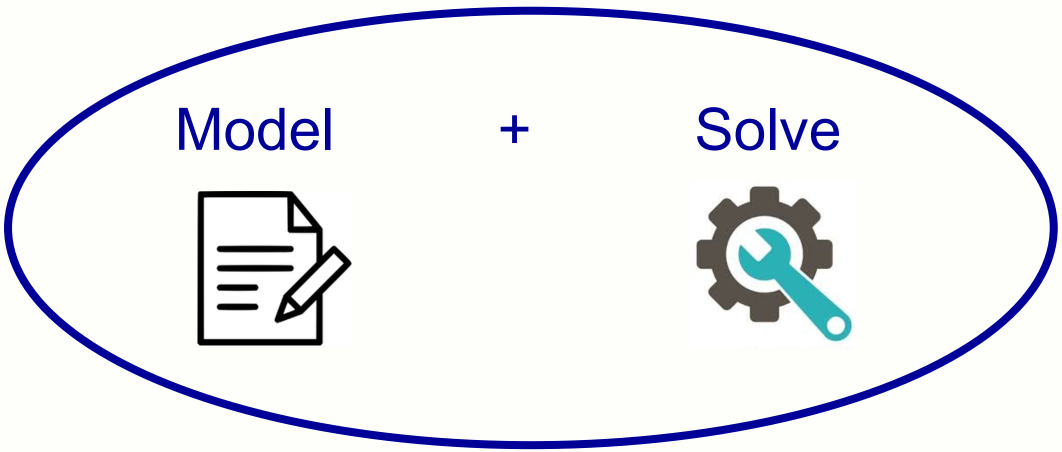
[CPMpy+Pandas+Geopy+Plotly demo]

Job shop scheduling

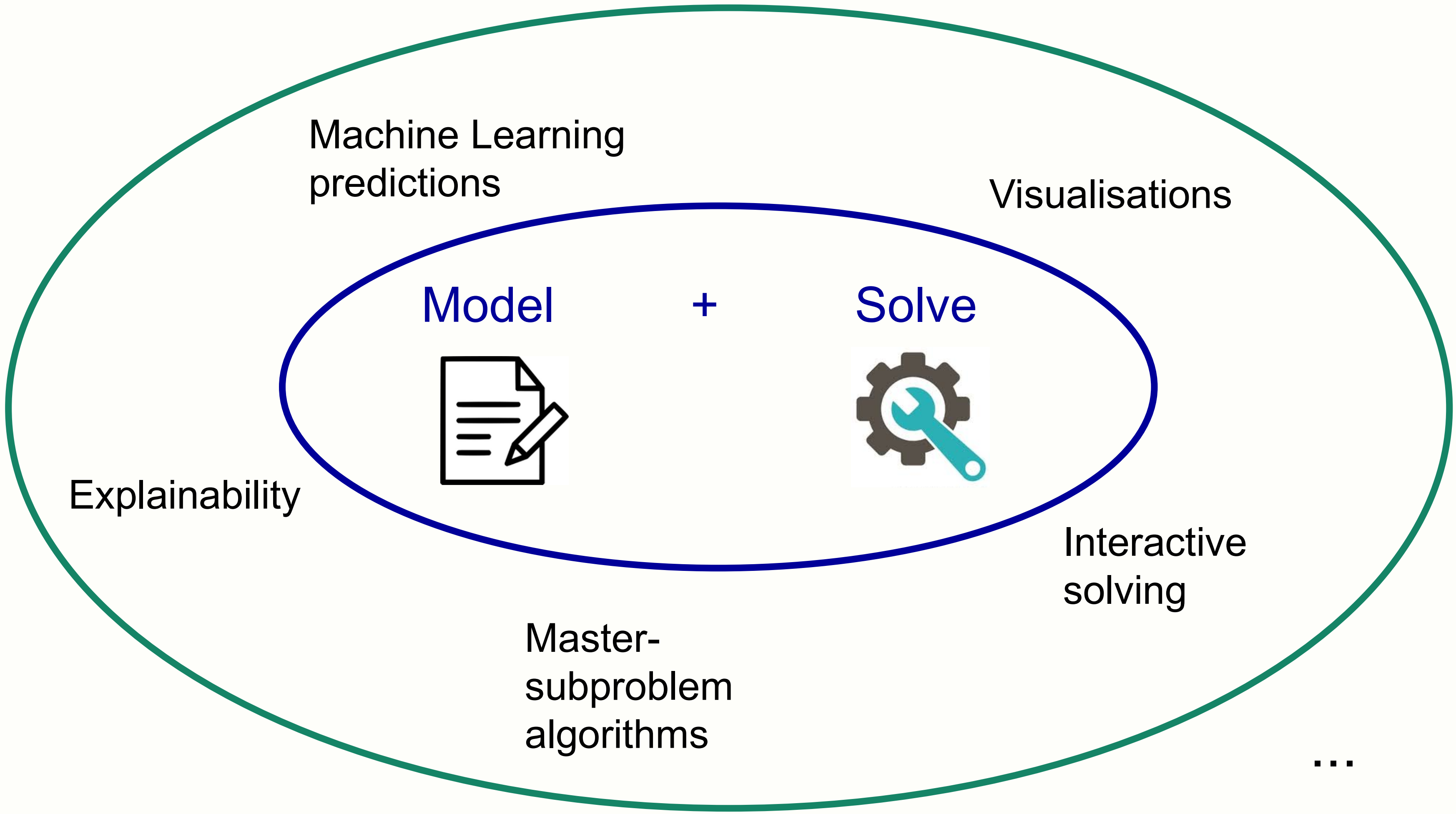
CPMpy+Pandas+Plotly demo

Beyond Model + Solve

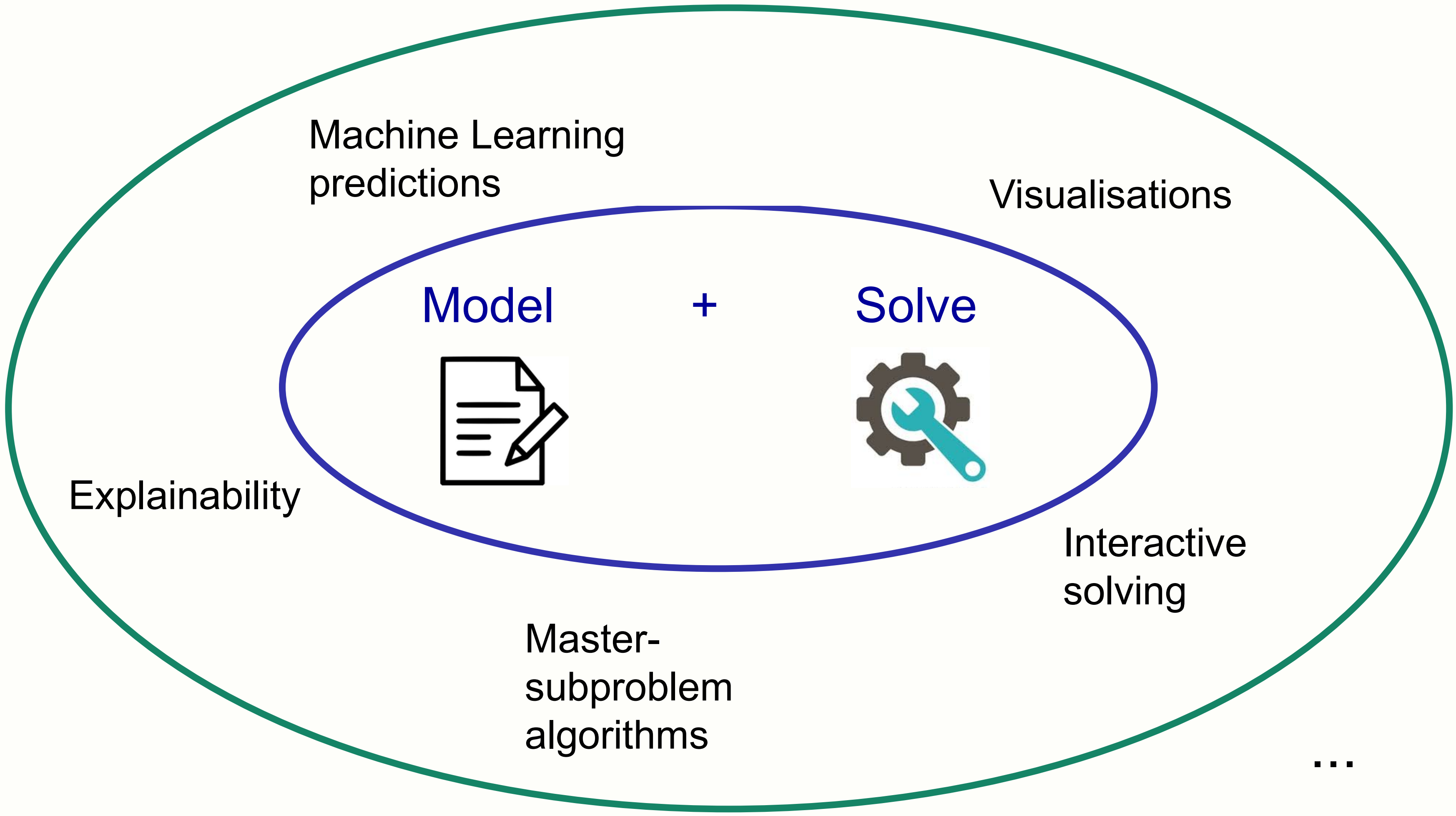
Wider view



Wider view: integration



Modern Constraint Solving



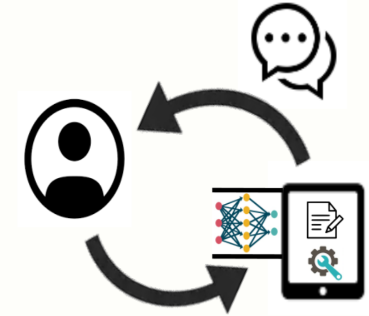
The changing role of solvers

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

I think we reached it… MiniZinc, Essence’

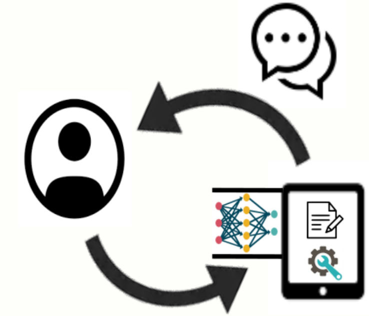
“Beyond NP” → constraint solving as an **oracle**

- 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?

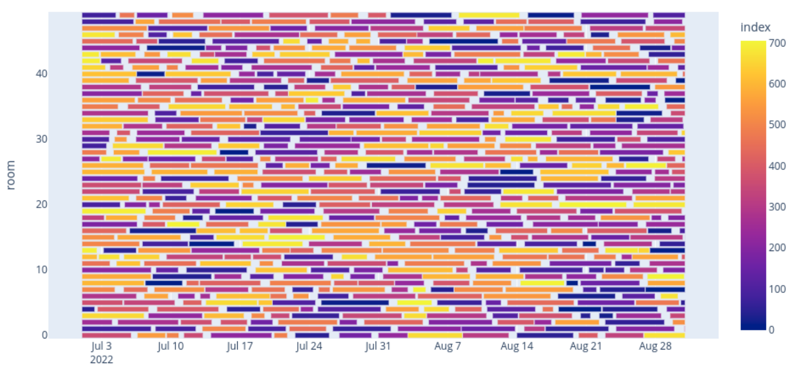
- **Efficient repeated solving**
  => Incremental

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

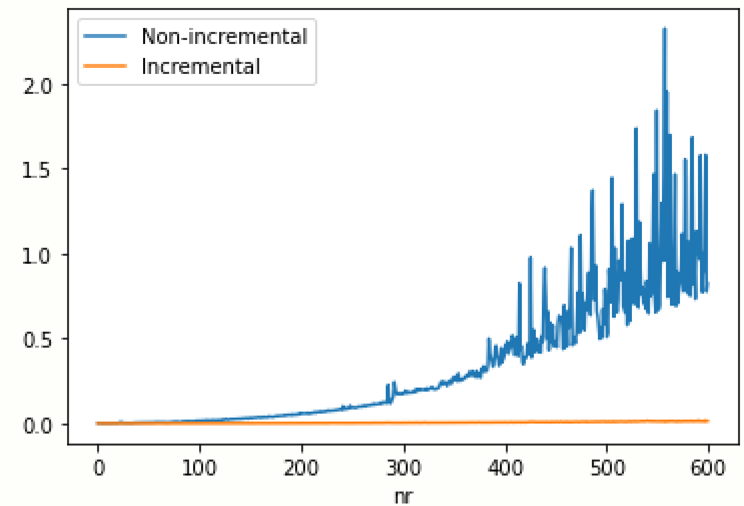
- Easy integration with Machine Learning libraries
  => Python and numpy arrays

Incremental room assignment problem

```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)
```



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



Incrementality

Solving:

- <u>MIP</u>: can add constraints, change objective
  (mechanisms not documented, e.g. start from previous basis)
- <u>SAT</u>: *assumption* variables: can be toggled on/off when calling solve
  (reuses learned clauses, variable activity)
- <u>CP</u>: if CP-SAT, assumption variables like SAT
- <u>SMT</u>: 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:

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

```
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
```

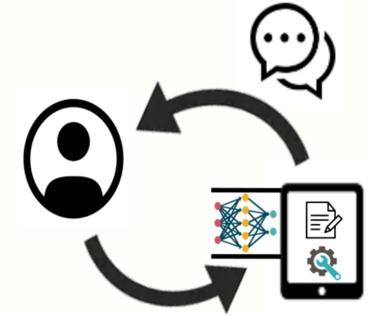
```
[3 0]
[3 1]
[3 2]
[2 0]
[1 0]
[2 1]
```

Adds constraint to model (even if already solved before)

Non-dominated solutions (disjunctive method)

```
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]



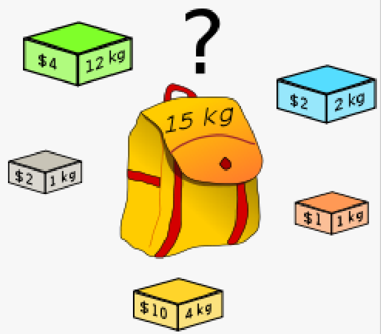
What would the ideal CP system be?

- 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

```
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("\tGurobi'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]

Gurobi: ExitStatus.OPTIMAL (0.0003108978271484375 seconds) : 32.0 [ True False  True False  True  True  True  Tru
e]
        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_l * w_l)
        minimize=sum(x_l * w_l for x_l, w_l 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

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

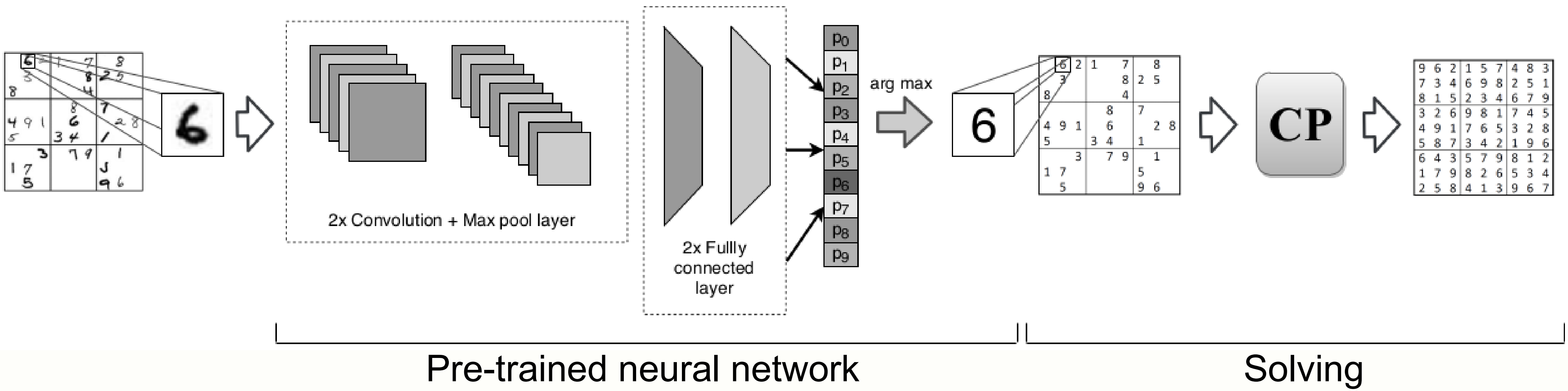
Tias Guns, Milan Pesa, Maxime Mulamba,
Ignace Bleukx, Emilio Gamba, Senne Berden

WWW.FLANDERSAIRESEARCH.BE

GET IT ON Google Play

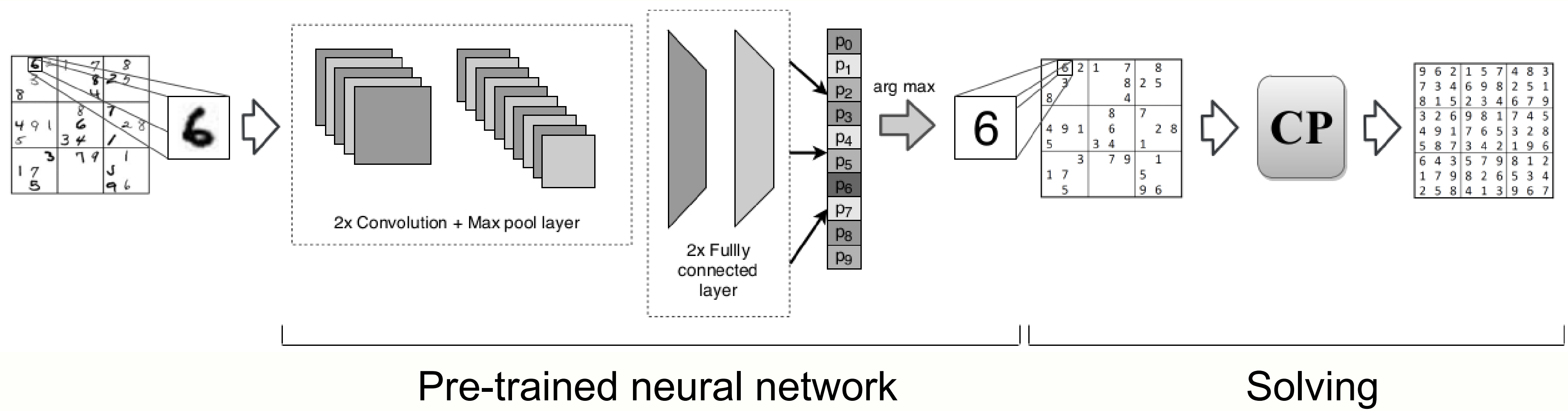
Perception-based constraint solving

Pedagogical instantiation: visual sudoku (naïve)



	accuracy img	accuracy cell	accuracy grid	failure rate grid	time average (s)
baseline	94.75%	15.51%	14.67%	84.43%	0.01

Perception-based constraint solving



<u>What is going on?</u>

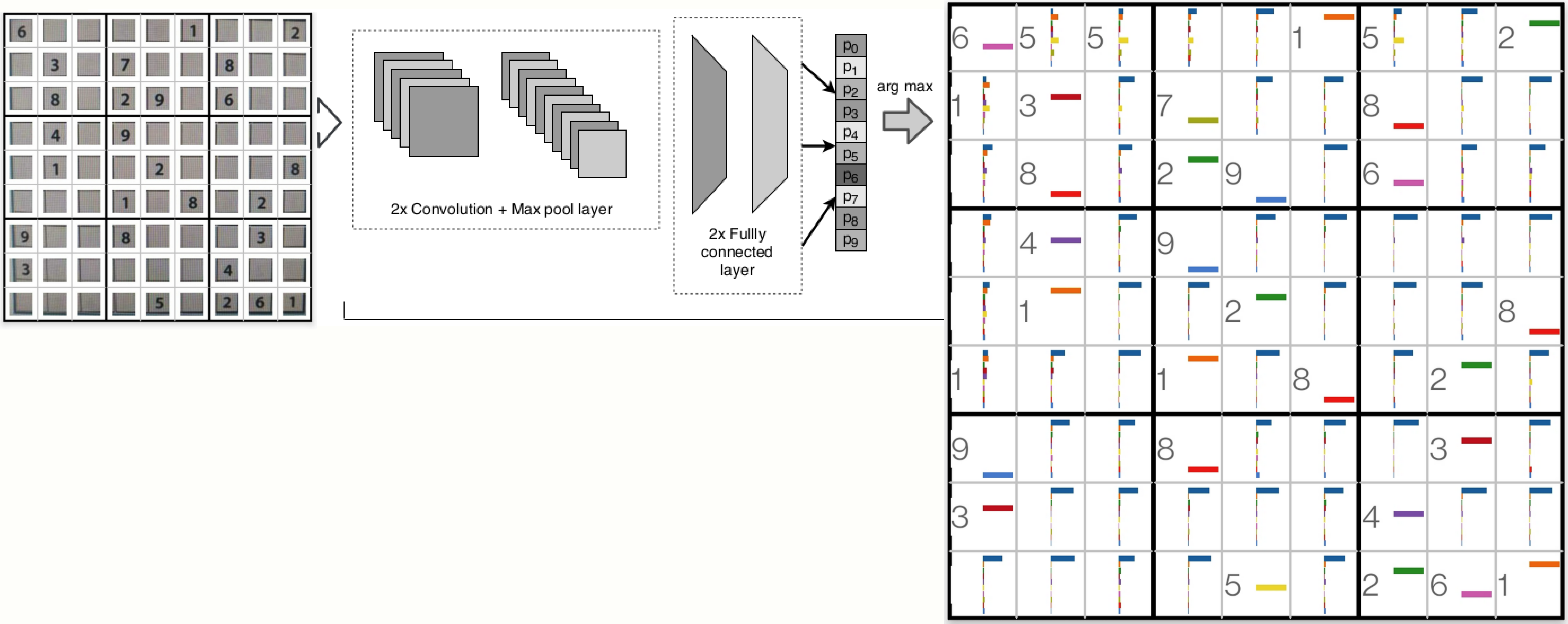
- Each cell predicts the maximum likelihood value:

$$\hat{y}_{ij} = \arg\max P(y_{ij} = k | X_{ij})$$

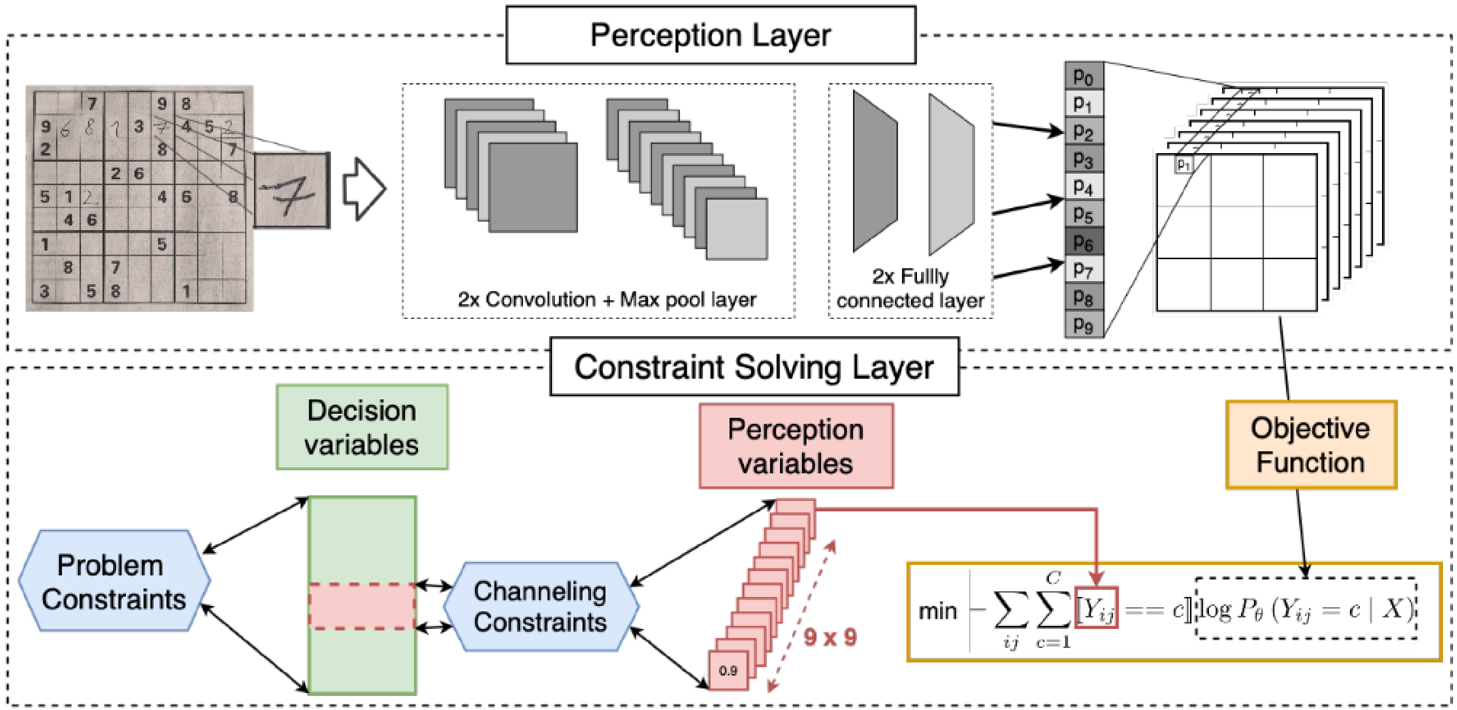
- But you need all 81 predictions (one for each given cell), it is a multi-output problem: together this is the ‘maximum likelihood’ interpretation

- If $\mathrm{sudoku}(\hat{y})$ = *False*: no solution, interpretation is wrong...

Perception-based constraint solving



What about the *next* most likely interpretation?



- Treat prediction as *joint inference* problem:

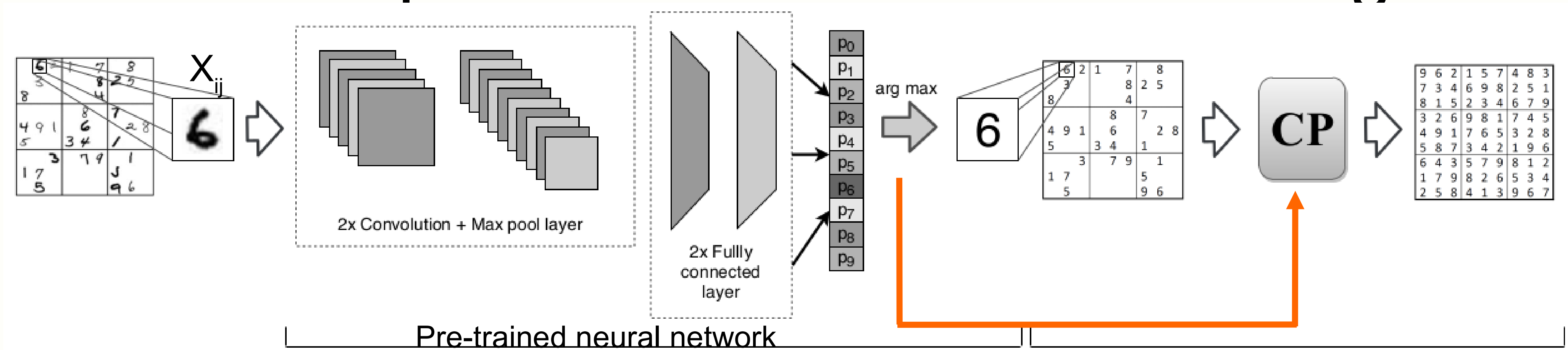
$$\hat{y} = \arg\max \prod_{ij} P(y_{ij} = k | X_{ij}) \quad \text{s.t.} \quad \text{sudoku}(\hat{y})$$

- This is the **constrained** ‘maximum likelihood’ interpretation

=> Structured output prediction

Used e.g. in NLP: [Punyakanok, COLING04]

Perception-based constraint solving



<u>Can we use a constraint solver for that?</u>

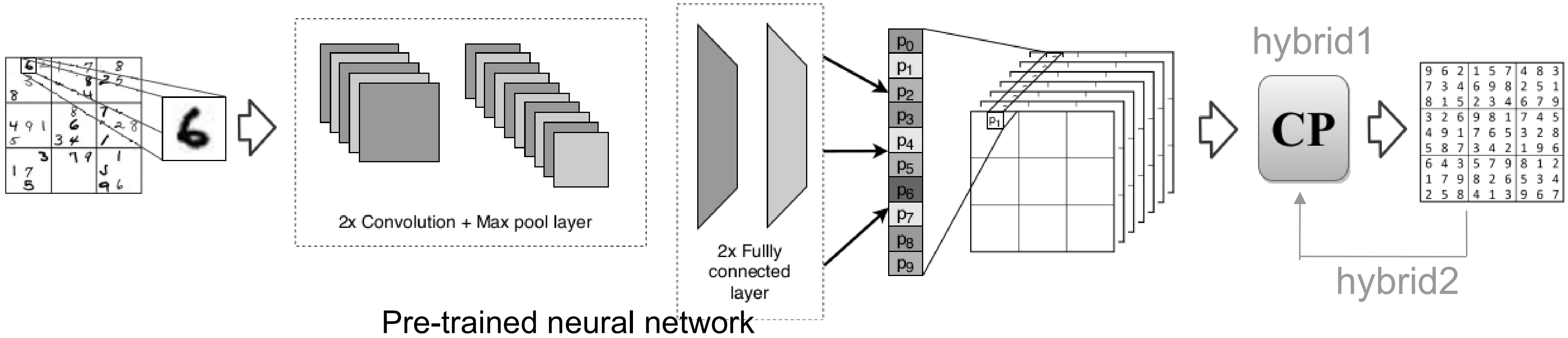
$$\hat{y} = \arg\max \prod_{ij} P(y_{ij} = k | X_{ij}) \quad \text{s.t.} \quad \text{sudoku}(\hat{y})$$

- Log-likelihood trick:

$$min \sum_{\substack{(i,j)\in \\ given}} \sum_{\substack{k\in \\ \{1,..,9\}}} \underbrace{-log(P_\theta(y_{ij} = k | X_{ij}))}_{constant} * \mathbb{1}[s_{ij} = k] \quad \text{s.t.} \quad \text{sudoku}(\hat{y})$$

Perception-based constraint solving

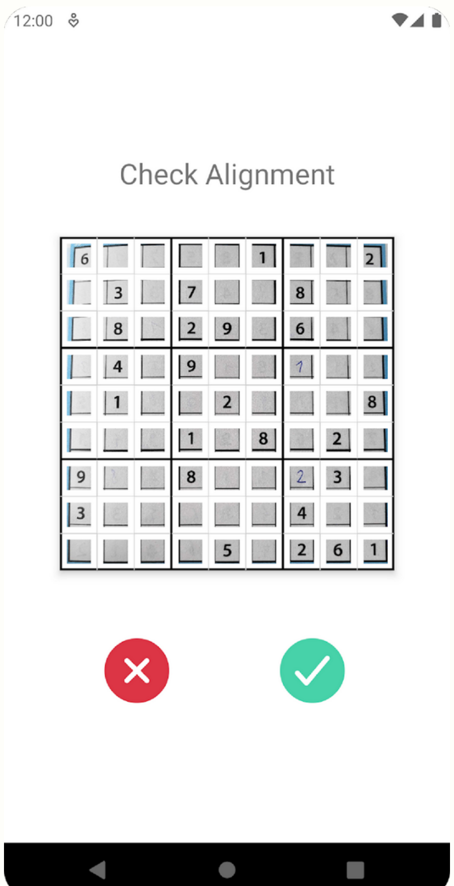
Hybrid: CP solver does *joint inference* over raw probabilities

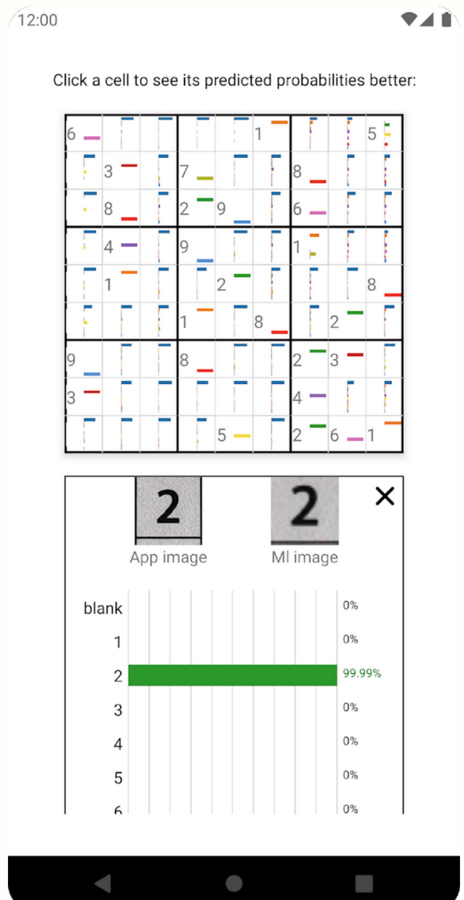


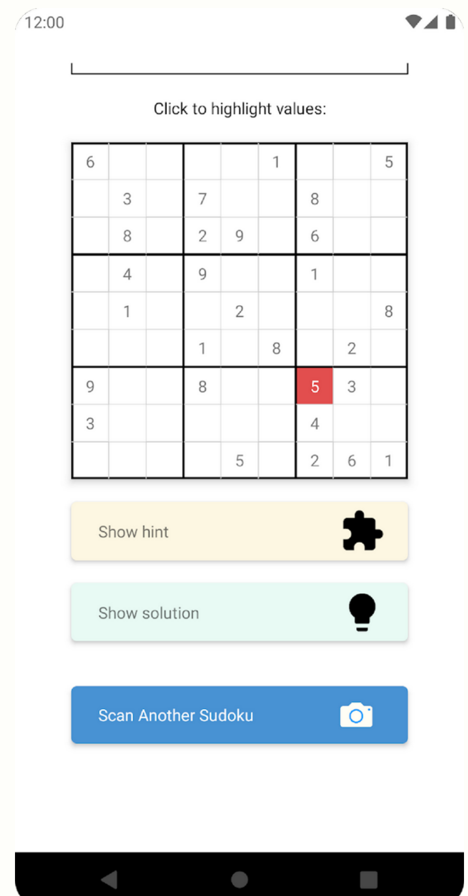
	accuracy img	cell	grid	failure rate grid	time average (s)
baseline	94.75%	15.51%	14.67%	84.43%	0.01
hybrid1	99.69%	99.38%	92.33%	0%	0.79
hybrid2	99.72%	99.44%	92.93%	0%	0.83

[Maxime Mulamba, Jayanta Mandi, Rocsildes Canoy, Tias Guns, CPAIOR20]

Sudoku Assistant demo, continued



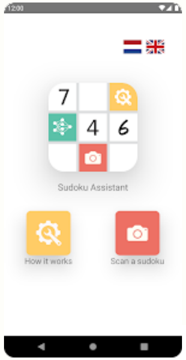




Implementation: integration

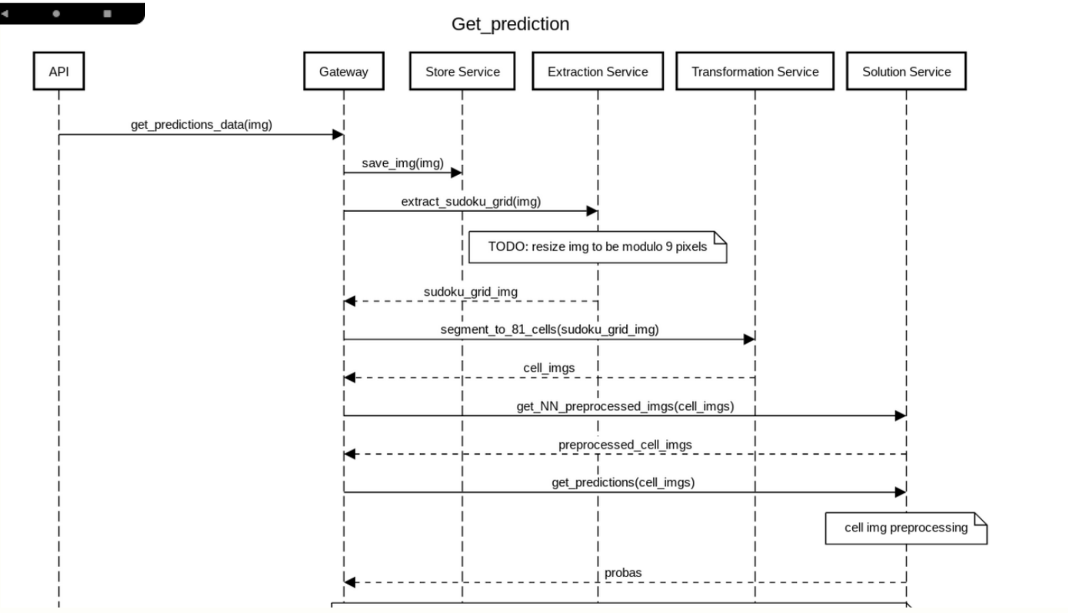
Frontend:

- React-native
- Only displays results



Backend:

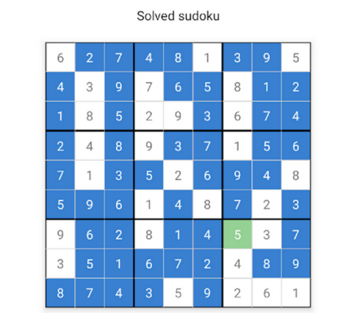
- FastAPI (Python)
- NN Service (PyTorch)
- Solver Service (CPMpy)
- Preloading, caching...



Show solution?

Trivial for CP system (subsecond),

Boring and demotivating for user?

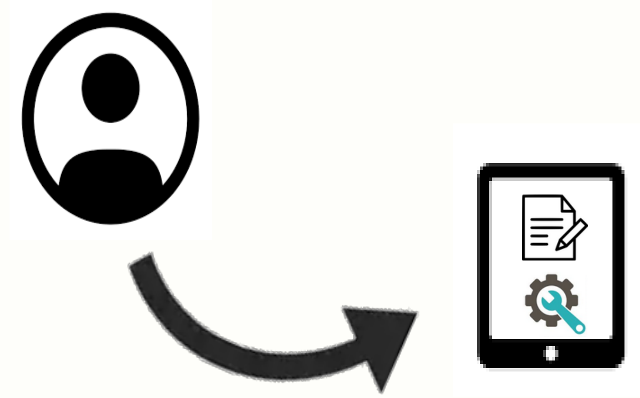


In general: <u>human-aware AI</u> & AI assistants:

- *Support* users in decision making
- Respect human *agency*
- Provide *explanations* and learning opportunities

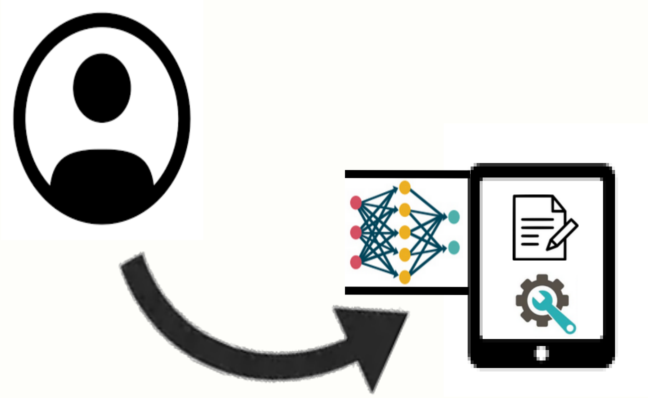
Constraint solving is more than mathematical abstractions...

Bigger picture



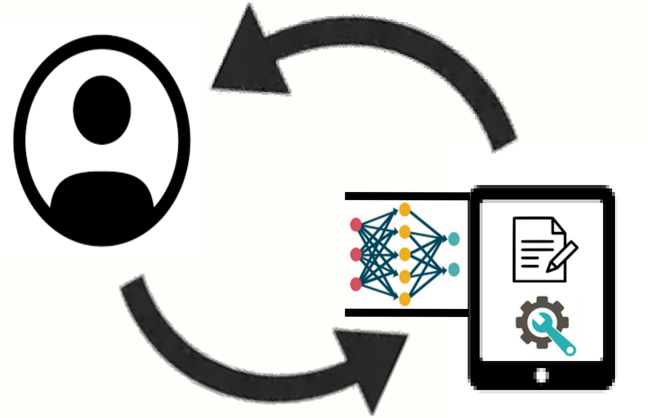
Bigger picture

- Learning implicit user preferences
- Learning from the environment



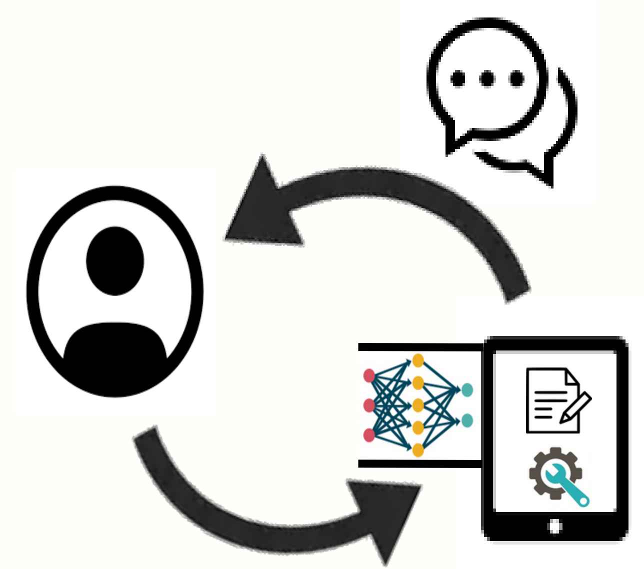
Bigger picture

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

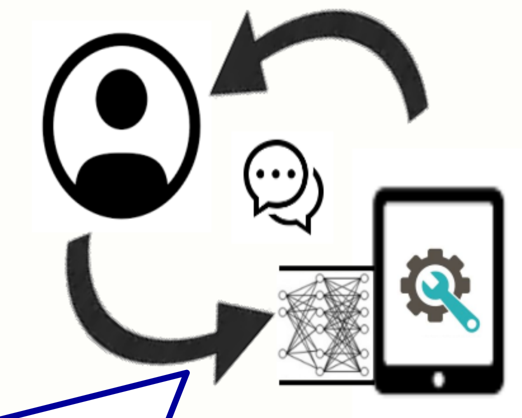


Bigger picture

- Learning implicit user preferences
- Learning from the environment
- Explaining constraint solving
- Stateful interaction



CHAT-Opt:
Conversational **H**uman-**A**ware **T**echnology for **Opt**imisation

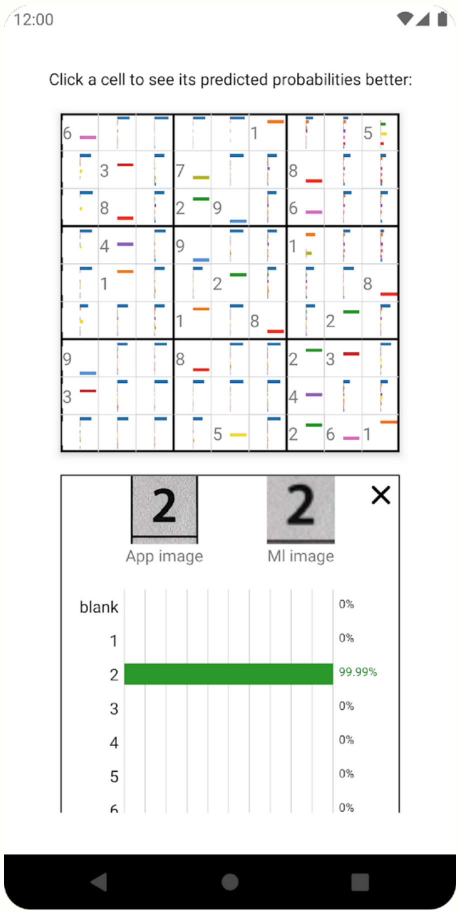


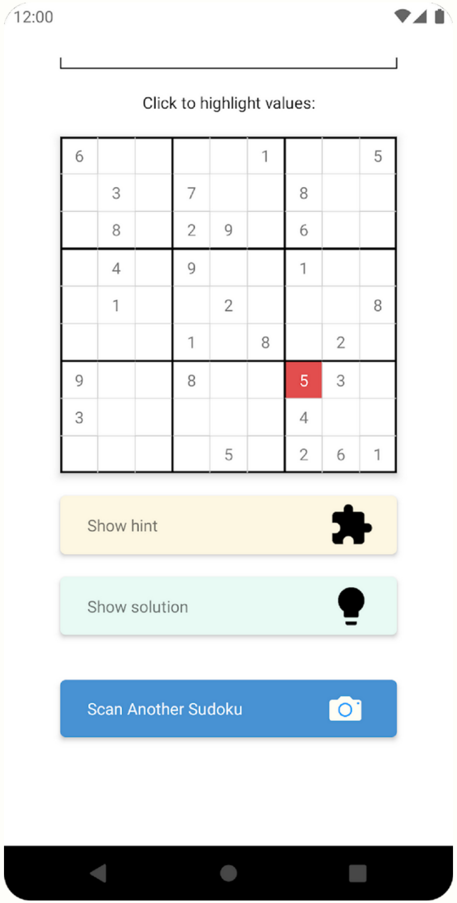
Towards **co-creation** of constraint optimisation solutions

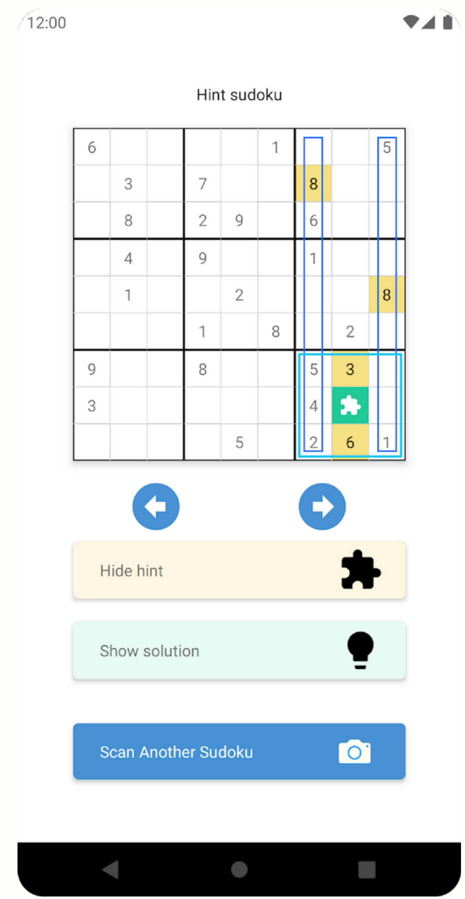
- Solver that learns from user and environment
- Towards conversational: explanations and stateful interaction

GET IT ON Google Play

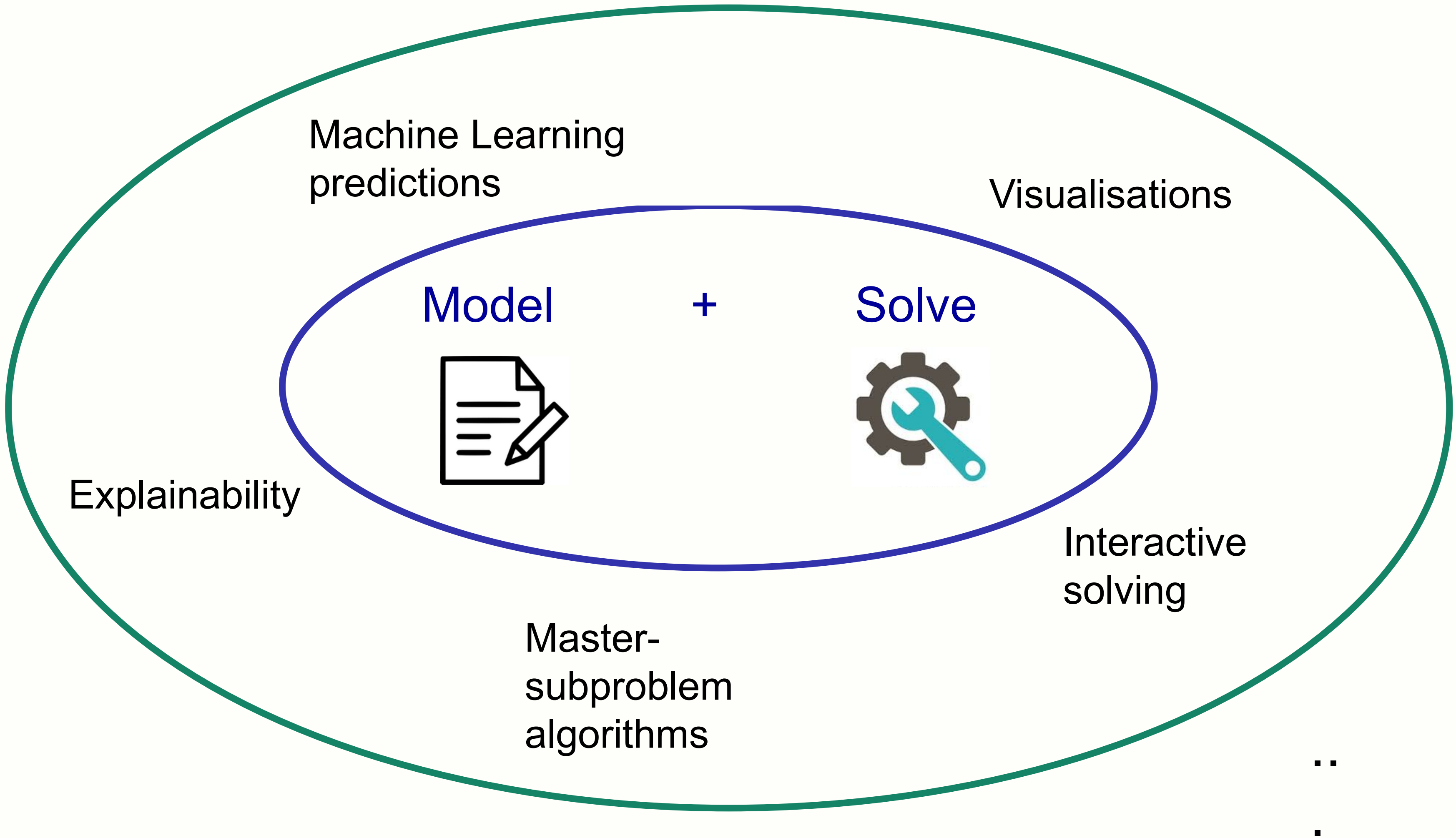
Sudoku Assistant, explanation steps



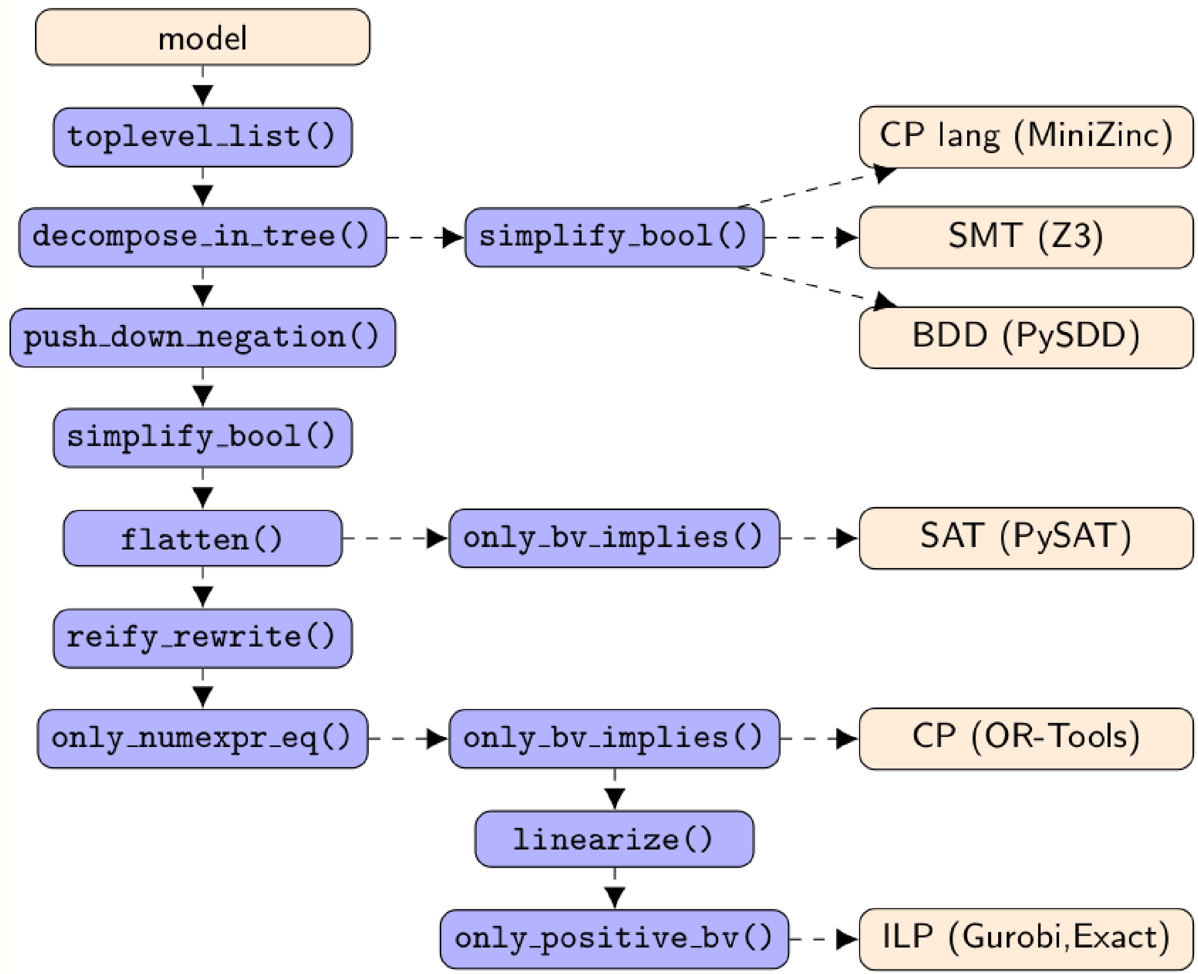




Modern Constraint Solving



CPMpy transformations in a nutshell



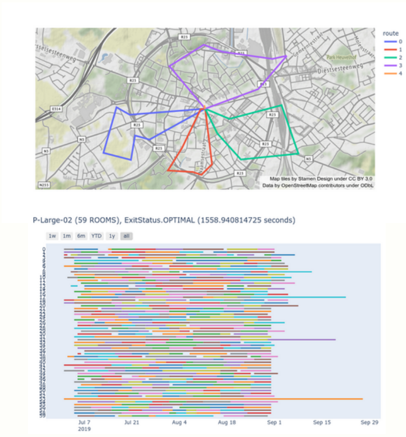
Solving Paradigms

	Model 	Solve 
SAT	Clauses (no objective)	Unit Propagation Clause learning
CP	Clauses, Linear, Global, Linear/any objective	Constraint Propagation= domain eliminations
MIP	Linear constraints, Linear objective	Relaxation Cutting planes

The end / to be continued

Model + Solve

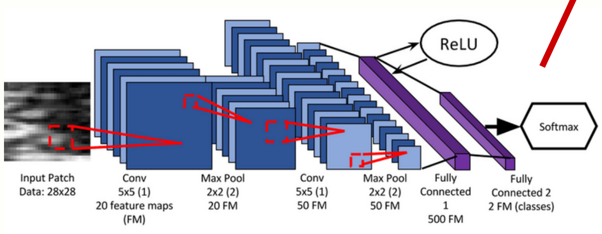
Decision variables
Constraints
Objective function



1) learn to model

2) learn to solve (faster)

Can we *learn* it instead?



Enjoy!

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