Explanation techniques for CP

Prof. Tias Guns, KU Leuven, Belgium

In collaboration with Ignace Bleukx, Emilio Gamba, Bart Bogaerts, Jo Devriendt, Dimos Tsouros

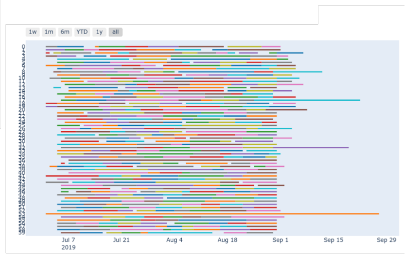
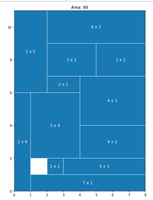
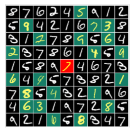
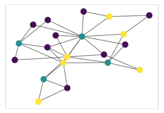
This presentation is an executable Jupyter notebook

Link to slides and more examples: https://github.com/CPMpy/XCP-explain

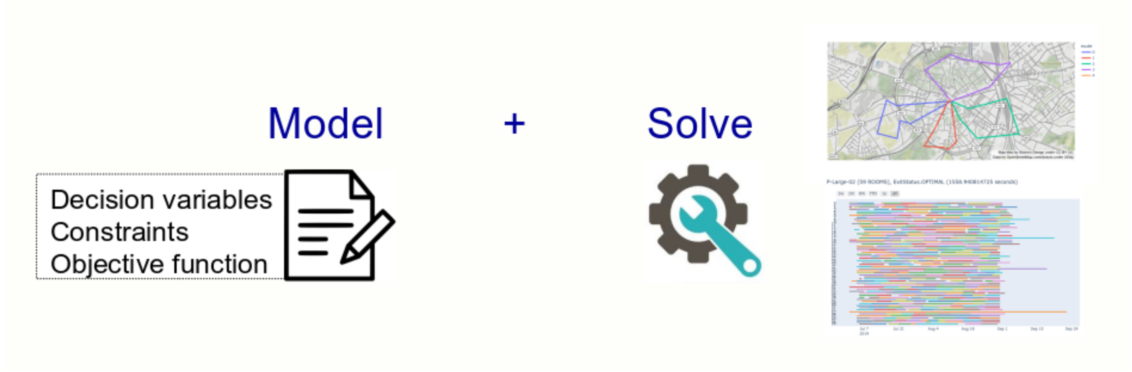
Constraint Solving

Solving combinatorial optimization problems in AI

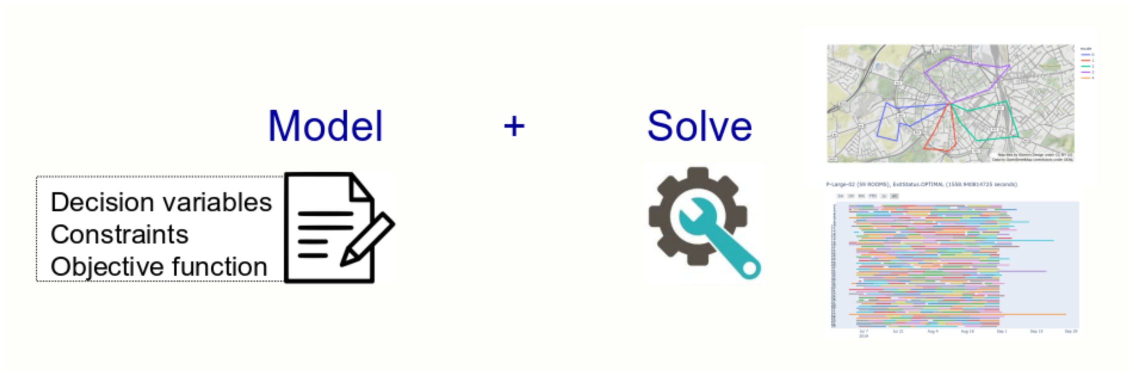
- Vehicle Routing
- Scheduling
- Manufacturing
- Other combinatorial problems ...

Model + Solve

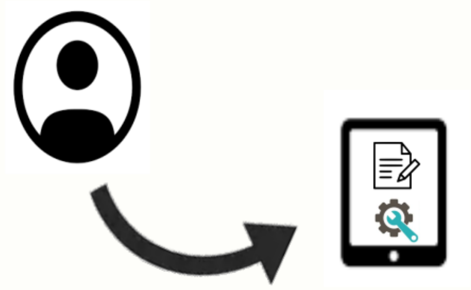

Model
+
Solve
Decision variables
Constraints
Objective function

Model + Solve



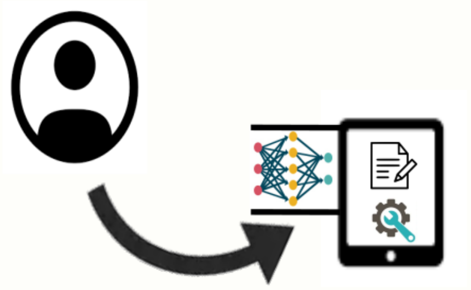
- What if no solution is found?
- What if the user does not *like* the solution?
- What if the user *expected* a different solution?
- ...

Bigger picture



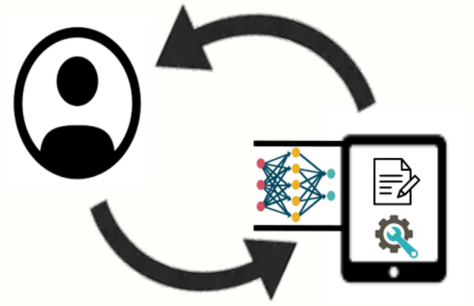
Bigger picture

- Learning from the environment
- Learning from the user



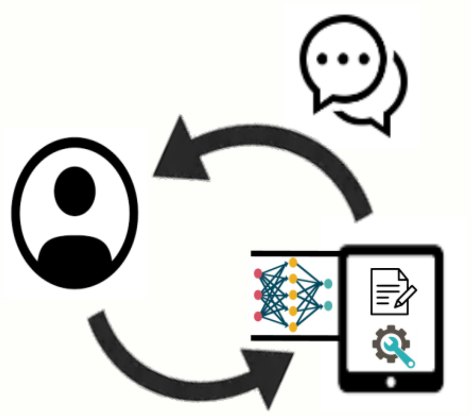
Bigger picture

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



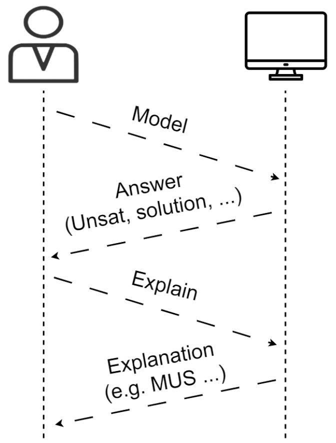
CHAT-Opt project

- Learning from the environment
- Learning from the user
- Explaining constraint solving
- Conversational, stateful interaction



Explainable Constraint Programming (XCP)

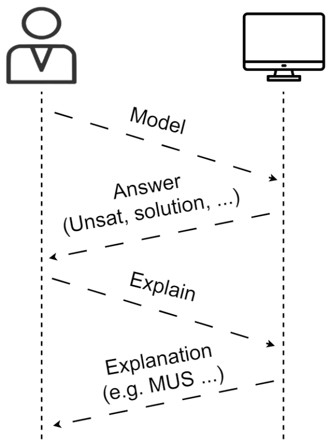
In general, "**Why X?**" (with X a solution or UNSAT)



Explainable Constraint Programming (XCP)

In general, "**Why X?**"   (with X a solution or UNSAT)

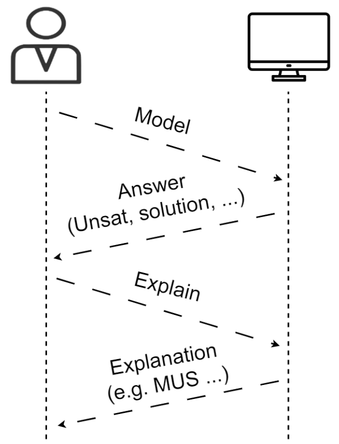
- **Deductive explanation:**
  - *What causes X?*
- **Counterfactual explanation**:
  - *What if I want Y instead of X?*



Explainable Constraint Programming (XCP)

In general, "**Why X?**" (with X a solution or UNSAT)

- **Deductive explanation:**
  - *What causes X?*
- **Counterfactual explanation**:
  - *What if I want Y instead of X?*



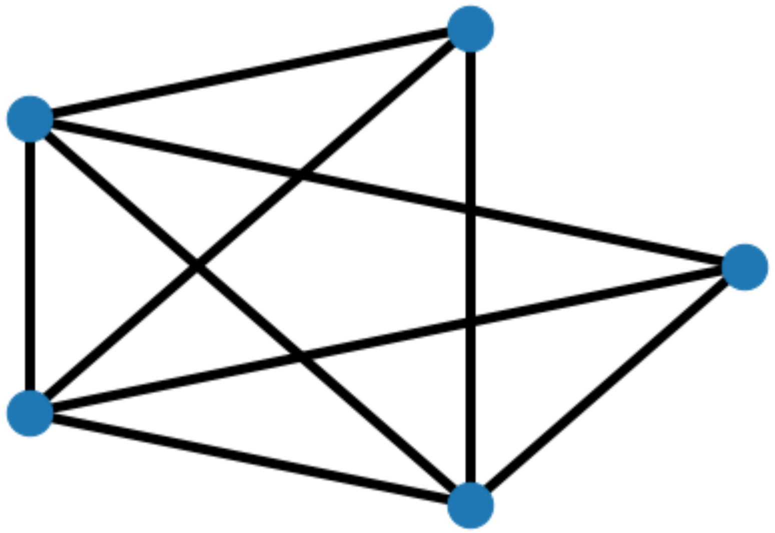
Note *explanations* also used in the context of lazy-clause generation: one propagator explains its inference to a SAT solver. We focus on **user-oriented explanations** involving multiple constraints.

Example XCP interaction

Toy example, graph coloring:

color each node such that no two adjacent nodes have the same color
(real example: assign each booking request (node) to a room (color) such that no temporally overlapping requests use the same room)

```
In [2]: G = nx.fast_gnp_random_graph(5, 0.8, seed=0)
        draw(G)
```



Example XCP interaction

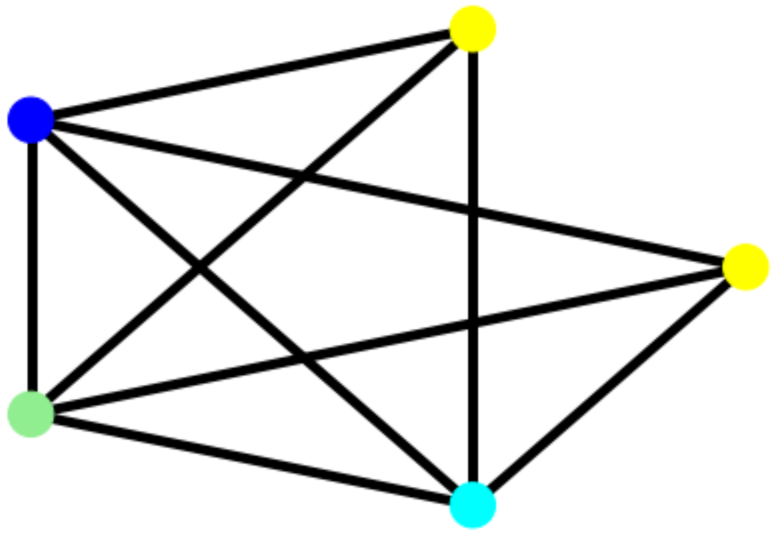
Lets color this graph...

Example XCP interaction

Lets color this graph...

```
In [3]:
m, nodes = graph_coloring(G, max_colors=None)
if m.solve():
    print(m.status())
    print(f"Found optimal coloring with {m.objective_value()} colors")
    draw(G, node_color=[cmap[n.value()] for n in nodes])
else:
    print("No solution found.")
```

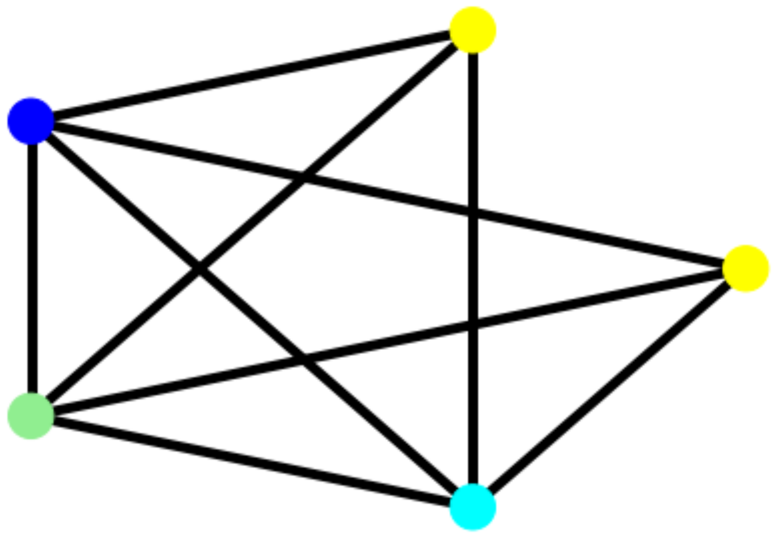
```
ExitStatus.OPTIMAL (0.007055108 seconds)
Found optimal coloring with 4 colors
```



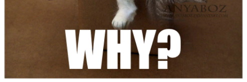
Example XCP interaction

```
In [4]: print(f"Found optimal coloring with {m.objective_value()} colors")
        draw(G, node_color=[cmap[n.value()] for n in nodes])
```

```
Found optimal coloring with 4 colors
```



yes... but why do we need 4?


NYABOZ
WHY?

Example XCP interaction

Why do we need 4 colors?

Deductive explanation: pinpoint to constraints *causing* this fact

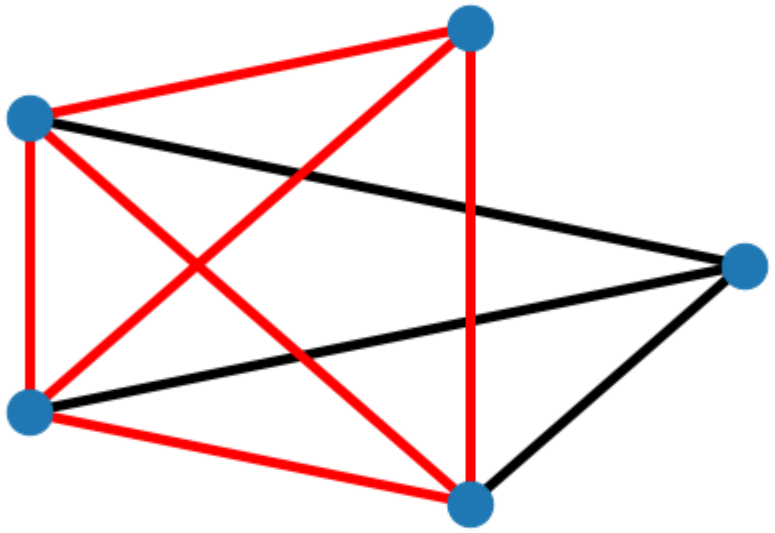
Example XCP interaction

Why do we need 4 colors?

Deductive explanation: pinpoint to constraints *causing* this fact

```
In [5]: m, nodes = graph_coloring(G, max_colors=3) # less than 4?
        if m.solve() is False:
            conflict = cpmpy.tools.explain.mus(m.constraints)  # Minimal Unsatis
            print("UNSAT is caused by the following constraints:")
            graph_highlight(G, conflict)
```

```
UNSAT is caused by the following constraints:
```



Example XCP interaction

Why do we need 4 colors?

Counterfactual explanation: pinpoint to constraint *changes* that would allow, e.g. 3 colors

Example XCP interaction

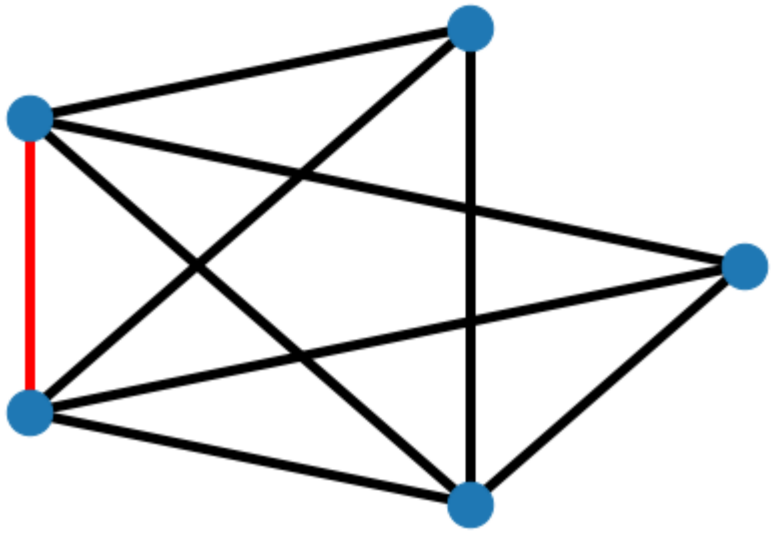
Why do we need 4 colors?

Counterfactual explanation: pinpoint to constraint *changes* that would allow, e.g. 3 colors

In [6]:

```
m, nodes = graph_coloring(G, max_colors=3) # less than 4?
if m.solve() is False:
    corr = cpmpy.tools.explain.mcs(m.constraints)  # Minimal Correction
    print("UNSAT can be resolved by removing the following constraints:'
    graph_highlight(G, corr)
```

```
UNSAT can be resolved by removing the following constraints:
```



Example XCP interaction

Why do we need 4 colors?

Counterfactual explanation: pinpoint to constraint *changes* that would allow, e.g. 3 colors

Can now compute the counterfactual solution:

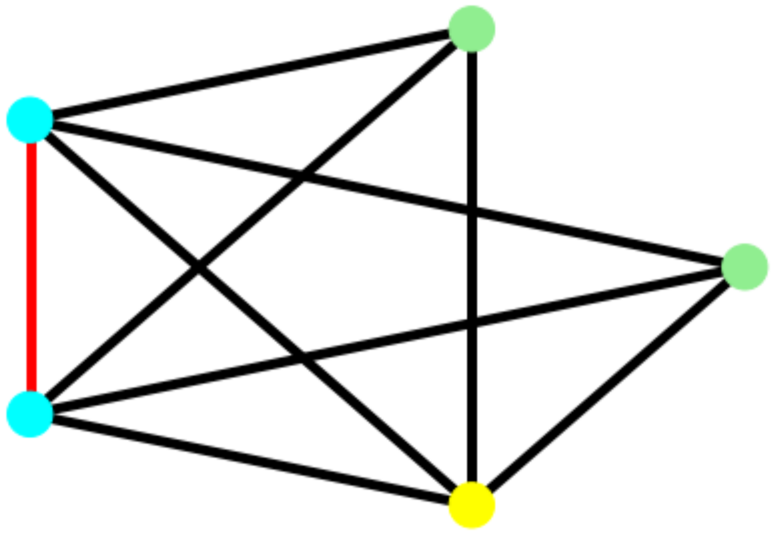
Example XCP interaction

Why do we need 4 colors?

Counterfactual explanation: pinpoint to constraint *changes* that would allow, e.g. 3 colors

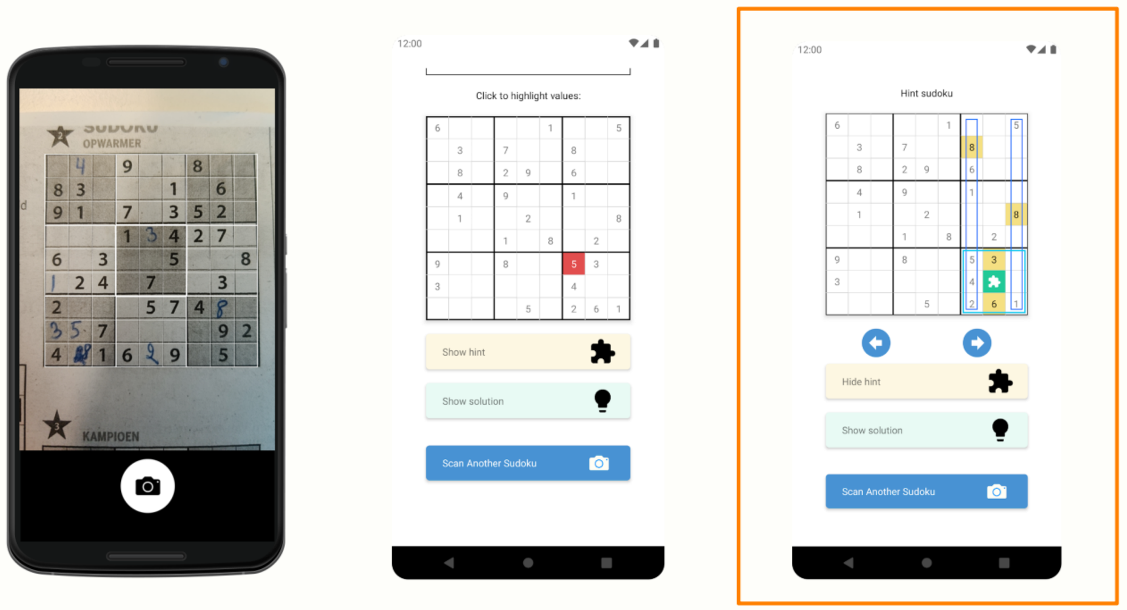
Can now compute the counterfactual solution:

```
In [7]: # compute and visualise counter-factual solution
        m2 = cp.Model([c for c in m.constraints if c not in corr])
        m2.solve()
        graph_highlight(G, corr, node_color=[cmap[n.value()] for n in nodes])
```



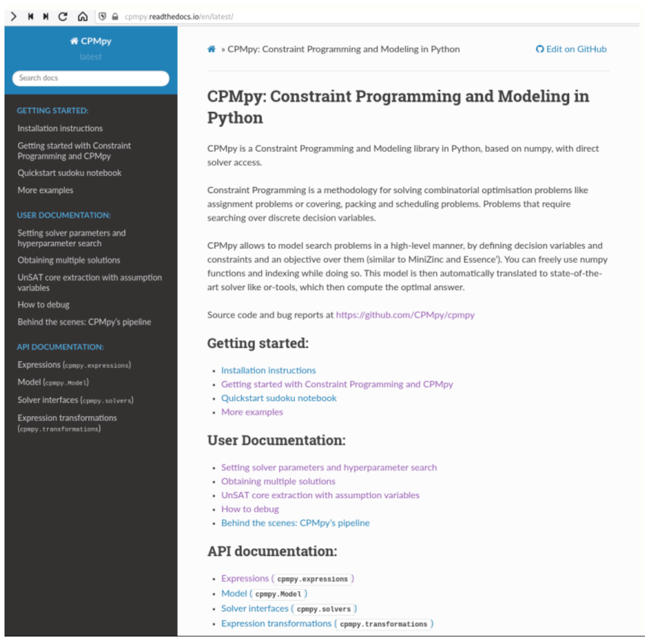
Explanation techniques in the wild

Sudoku Assistant, explanation steps



CPMpy: http://cpmpy.readthedocs.io

We will use the CPMpy modeling library in Python for this presentation


cpmpy.readthedocs.io/en/latest/
CPMpy
latest
Search docs
GETTING STARTED:
Installation instructions
Getting started with Constraint Programming and CPMpy
Quickstart sudoku notebook
More examples
USER DOCUMENTATION:
Setting solver parameters and hyperparameter search
Obtaining multiple solutions
UnSAT core extraction with assumption variables
How to debug
Behind the scenes: CPMpy's pipeline
API DOCUMENTATION:
Expressions (cpmpy.expressions)
Model (cpmpy.Model)
Solver interfaces (cpmpy.solvers)
Expression transformations (cpmpy.transformations)
» CPMpy: Constraint Programming and Modeling in Python
Edit on GitHub

CPMpy: Constraint Programming and Modeling in Python

CPMpy is a Constraint Programming and Modeling library in Python, based on numpy, with direct solver access.

Constraint Programming is a methodology for solving combinatorial optimisation problems like assignment problems or covering, packing and scheduling problems. Problems that require searching over discrete decision variables.

CPMpy allows to model search problems in a high-level manner, by defining decision variables and constraints and an objective over them (similar to MiniZinc and Essence'). You can freely use numpy functions and indexing while doing so. This model is then automatically translated to state-of-the-art solver like or-tools, which then compute the optimal answer.

Source code and bug reports at https://github.com/CPMpy/cpmpy

Getting started:

- Installation instructions
- Getting started with Constraint Programming and CPMpy
- Quickstart sudoku notebook
- More examples

User Documentation:

- Setting solver parameters and hyperparameter search
- Obtaining multiple solutions
- UnSAT core extraction with assumption variables
- How to debug
- Behind the scenes: CPMpy's pipeline

API documentation:

- Expressions (`cpmpy.expressions`)
- Model (`cpmpy.Model`)
- Solver interfaces (`cpmpy.solvers`)
- Expression transformations (`cpmpy.transformations`)

https://github.com/CPMpy/cpmpy

- Open source
- Python/Numpy based
- Direct, incremental solver access

Supported solvers:

- ORTools (CP)
- Exact (PseudoBoolean+int)
- Gurobi (ILP)
- Z3 (SMT)
- PySAT, PySDD (SAT, know. comp.)
- More to come… (Choco, SCIP, CP Opt)

Running example in this talk: Nurse Scheduling

- The assignment of *shifts* and *holidays* to nurses.
- Each nurse has their own restrictions and preferences, as does the hospital.

In [9]:
```
#instance = "http://www.schedulingbenchmarks.org/nrp/data/Instance1.txt'
instance = "Benchmarks/Instance1.txt"
data = get_data(instance)

factory = NurseSchedulingFactory(data)
model, nurse_view = factory.get_full_model()  # CPMpy model with all cor

model.solve()
visualize(nurse_view.value(), factory)  # live decorated dataframe
```

Out[9]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan	F	D	D	D	D	F	F	D	D	F	F	D	D
Katherine	D	D	D	D	D	F	F	D	D	F	F	F	D
Robert	D	D	D	F	F	D	D	F	F	D	D	D	F
Jonathan	D	D	F	F	F	D	D	D	D	D	F	F	F
William	F	D	D	D	D	F	F	D	D	F	F	D	D
Richard	D	D	D	F	F	F	F	D	D	D	F	F	D
Kristen	F	F	D	D	D	F	F	D	D	F	F	D	D
Kevin	D	D	F	F	D	D	F	F	D	D	D	D	F
Cover D	5/5	7/7	6/6	4/4	5/5	3/5	2/5	6/6	7/7	4/4	2/2	5/5	5/6

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets ⏪
  - efficient MUSes
  - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists

Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function

Deductive Explanations for UNSAT problems

```
In [11]: # decision model, add all nurse preferences as hard constraints
         factory = NurseSchedulingFactory(data)
         model, nurse_view = factory.get_decision_model()
         model.solve()

Out[11]: False
```

Deductive Explanations for UNSAT problems

```
In [11]: # decision model, add all nurse preferences as hard constraints
         factory = NurseSchedulingFactory(data)
         model, nurse_view = factory.get_decision_model()
         model.solve()

Out[11]: False
```

... no solution found

Deductive Explanations for UNSAT problems

```
In [11]: # decision model, add all nurse preferences as hard constraints
         factory = NurseSchedulingFactory(data)
         model, nurse_view = factory.get_decision_model()
         model.solve()

Out[11]:  False
```

... no solution found

```
In [12]: constraints = toplevel_list(model.constraints, merge_and=False) # normal
         print(f"Model has {len(constraints)} constraints:")
         for cons in constraints: print("-", cons)
```

```
Model has 168 constraints:
- Megan cannot work more than 14 shifts of type 1
- Katherine cannot work more than 14 shifts of type 1
- Robert cannot work more than 14 shifts of type 1
- Jonathan cannot work more than 14 shifts of type 1
- William cannot work more than 14 shifts of type 1
- Richard cannot work more than 14 shifts of type 1
- Kristen cannot work more than 14 shifts of type 1
- Kevin cannot work more than 14 shifts of type 1
- Megan cannot work more than 4320min
- Katherine cannot work more than 4320min
- Robert cannot work more than 4320min
- Jonathan cannot work more than 4320min
- William cannot work more than 4320min
- Richard cannot work more than 4320min
- Kristen cannot work more than 4320min
- Kevin cannot work more than 4320min
- Megan cannot work more than 3360min
- Katherine cannot work more than 3360min
- Robert cannot work more than 3360min
- Jonathan cannot work more than 3360min
- William cannot work more than 3360min
- Richard cannot work more than 3360min
- Kristen cannot work more than 3360min
- Kevin cannot work more than 3360min
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
```

- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Megan can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Katherine can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Robert can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- Jonathan can work at most 5 days before having a day off
- William can work at most 5 days before having a day off

- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- William can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Richard can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kristen can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off

```
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Kevin can work at most 5 days before having a day off
- Megan should work at least 2 days before having a day off
- Katherine should work at least 2 days before having a day off
- Robert should work at least 2 days before having a day off
- Jonathan should work at least 2 days before having a day off
- William should work at least 2 days before having a day off
- Richard should work at least 2 days before having a day off
- Kristen should work at least 2 days before having a day off
- Kevin should work at least 2 days before having a day off
- Megan should work at most 1 weekends
- Katherine should work at most 1 weekends
- Robert should work at most 1 weekends
- Jonathan should work at most 1 weekends
- William should work at most 1 weekends
- Richard should work at most 1 weekends
- Kristen should work at most 1 weekends
- Kevin should work at most 1 weekends
- Megan has a day off on Mon 1
- Katherine has a day off on Sat 1
- Robert has a day off on Tue 2
- Jonathan has a day off on Wed 1
- William has a day off on Wed 2
- Richard has a day off on Sat 1
- Kristen has a day off on Tue 1
- Kevin has a day off on Mon 2
- Megan should have at least 2 consecutive days off
- Katherine should have at least 2 consecutive days off
- Robert should have at least 2 consecutive days off
```

- Jonathan should have at least 2 consecutive days off
- William should have at least 2 consecutive days off
- Richard should have at least 2 consecutive days off
- Kristen should have at least 2 consecutive days off
- Kevin should have at least 2 consecutive days off
- Megan requests to work shift D on Wed 1
- Megan requests to work shift D on Thu 1
- Katherine requests to work shift D on Mon 1
- Katherine requests to work shift D on Tue 1
- Katherine requests to work shift D on Wed 1
- Katherine requests to work shift D on Thu 1
- Katherine requests to work shift D on Fri 1
- Robert requests to work shift D on Mon 1
- Robert requests to work shift D on Tue 1
- Robert requests to work shift D on Wed 1
- Robert requests to work shift D on Thu 1
- Robert requests to work shift D on Fri 1
- Jonathan requests to work shift D on Tue 2
- Jonathan requests to work shift D on Wed 2
- Richard requests to work shift D on Mon 1
- Richard requests to work shift D on Tue 1
- Kevin requests to work shift D on Wed 2
- Kevin requests to work shift D on Thu 2
- Kevin requests to work shift D on Fri 2
- Kevin requests to work shift D on Sat 2
- Kevin requests to work shift D on Sun 2
- Robert requests to not work shift D on Sat 2
- Robert requests to not work shift D on Sun 2
- Richard requests to not work shift D on Tue 2
- Kevin requests to not work shift D on Wed 1
- Kevin requests to not work shift D on Thu 1
- Shift D on Mon 1 must be covered by 5 nurses out of 8

Deductive Explanations for UNSAT problems

The set of all constraints is unsatisfiable.

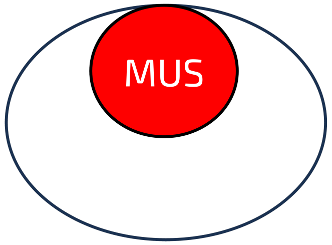
Deductive Explanations for UNSAT problems

The set of all constraints is unsatisfiable.

But do all constraints contribute to this?

Deductive Explanations for UNSAT problems

Minimal Unsatisfiable Subset (MUS)



Pinpoint to constraints causing a conflict

… trim model to a minimal set of constraints

… minimize cognitive burden for user

How to compute a MUS?

Deletion-based MUS algorithm

[Joao Marques-Silva. Minimal Unsatisfiability: Models, Algorithms and Applications. ISMVL 2010. pp. 9-14]

How to compute a MUS?

Deletion-based MUS algorithm

[Joao Marques-Silva. Minimal Unsatisfiability: Models, Algorithms and Applications. ISMVL 2010. pp. 9-14]

In [13]:
```python
def mus_naive(constraints):
    m = cp.Model(constraints)
    assert m.solve() is False, "Model should be UNSAT"

    core = constraints
    i = 0
    while i < len(core):
        subcore = core[:i] + core[i+1:]  # try all but constraint 'i'
        if cp.Model(subcore).solve() is True:
            i += 1  # removing 'i' makes it SAT, need to keep for UNSAT
        else:
            core = subcore #  can safely delete 'i'
    return core
```

How to compute a MUS, efficiently?

```
In [14]: t0 = time.time()
         core = mus_naive(constraints)
         print(f"Naive MUS took {time.time()-t0} seconds")
```

```
Naive MUS took 45.81037354469299 seconds
```

How to compute a MUS, efficiently?

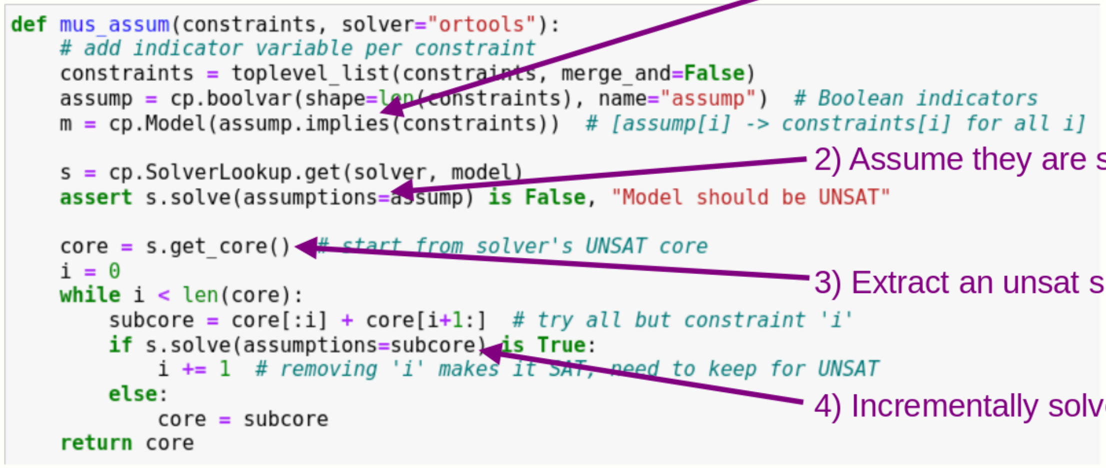
```
In [14]: t0 = time.time()
         core = mus_naive(constraints)
         print(f"Naive MUS took {time.time()-t0} seconds")

         Naive MUS took 45.81037354469299 seconds

In [15]: t0 = time.time()
         core = cpmpy.tools.explain.mus(constraints, solver="exact")
         print(f"Assumption-based MUS took {time.time()-t0} seconds")

         Assumption-based MUS took 2.8065266609191895 seconds
```

How to compute a MUS, <u>efficiently</u>?

How to compute a MUS, efficiently?

```
In [16]: def mus_assum(constraints, solver="ortools"):
             # add indicator variable per expression
             constraints = toplevel_list(constraints, merge_and=False)
             assump = cp.boolvar(shape=len(constraints), name="assump")  # Boolea
             m = cp.Model(assump.implies(constraints))  # [assump[i] -> constrai

             s = cp.SolverLookup.get(solver, model)
             assert s.solve(assumptions=assump) is False, "Model should be UNSAT"

             core = s.get_core()  # start from solver's UNSAT core of assumption
             i = 0
             while i < len(core):
                 subcore = core[:i] + core[i+1:]  # try all but constraint 'i'
                 if s.solve(assumptions=subcore) is True:
                     i += 1  # removing 'i' makes it SAT, need to keep for UNSAT
                 else:
                     core = subcore
             return [c for c,var in zip(constraints,assump) if var in core]
```

How to compute a MUS, efficiently?



```python
def mus_assum(constraints, solver="ortools"):
    # add indicator variable per constraint
    constraints = toplevel_list(constraints, merge_and=False)
    assump = cp.boolvar(shape=len(constraints), name="assump")  # Boolean indicators
    m = cp.Model(assump.implies(constraints))  # [assump[i] -> constraints[i] for all i]

    s = cp.SolverLookup.get(solver, model)
    assert s.solve(assumptions=assump) is False, "Model should be UNSAT"

    core = s.get_core() # start from solver's UNSAT core
    i = 0
    while i < len(core):
        subcore = core[:i] + core[i+1:]  # try all but constraint 'i'
        if s.solve(assumptions=subcore) is True:
            i += 1  # removing 'i' makes it SAT, need to keep for UNSAT
        else:
            core = subcore
    return core
```

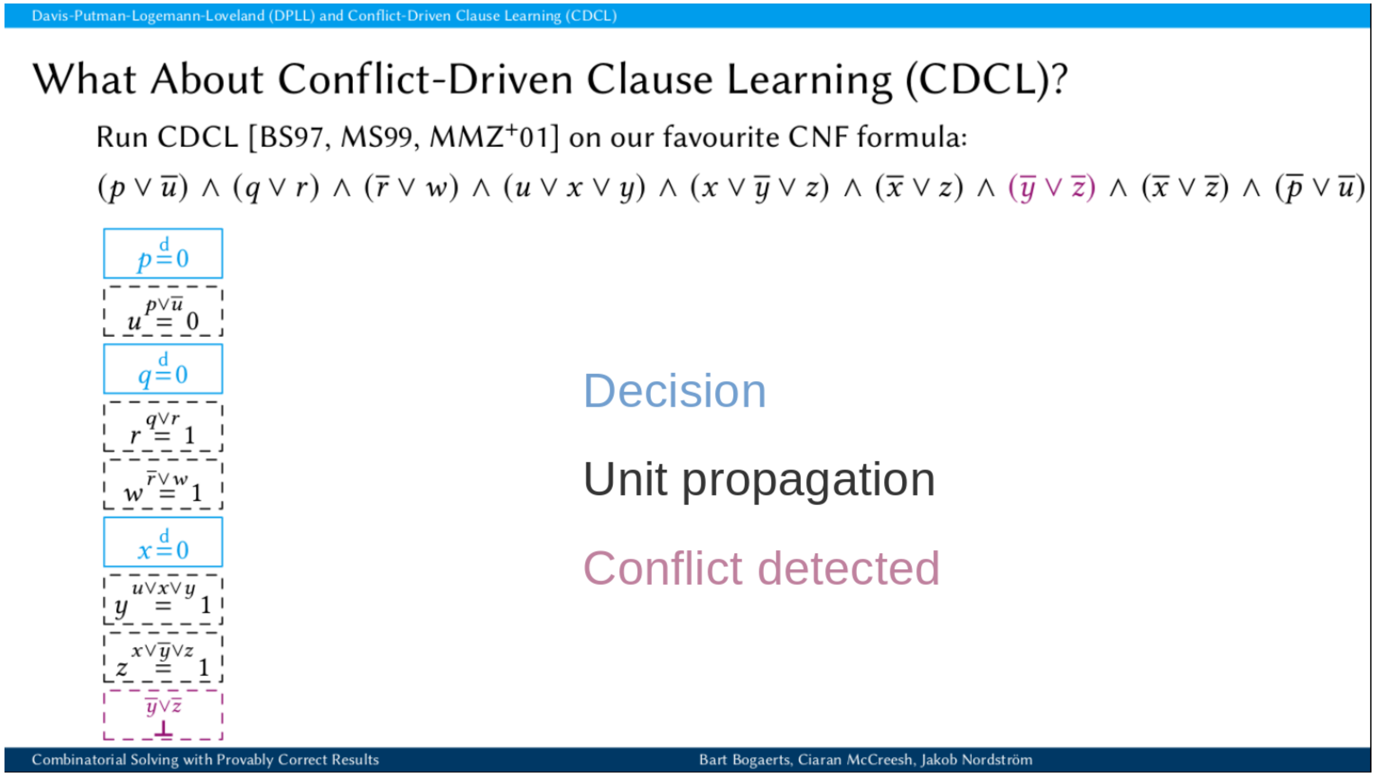
1) Add Boolean indicator variables

2) Assume they are set to ‘true’

3) Extract an unsat subset from solver

4) Incrementally solve diff. subsets

Deepdive: incremental CDCL solving with assumption variables 1/4



What About Conflict-Driven Clause Learning (CDCL)?

Run CDCL [BS97, MS99, MMZ+01] on our favourite CNF formula:

$$(p \lor \overline{u}) \land (q \lor r) \land (\overline{r} \lor w) \land (u \lor x \lor y) \land (x \lor \overline{y} \lor z) \land (\overline{x} \lor z) \land (\overline{y} \lor \overline{z}) \land (\overline{x} \lor \overline{z}) \land (\overline{p} \lor \overline{u})$$

$p \overset{d}{=} 0$

$u \overset{p \lor \overline{u}}{=} 0$

$q \overset{d}{=} 0$

$r \overset{q \lor r}{=} 1$

$w \overset{\overline{r} \lor w}{=} 1$

$x \overset{d}{=} 0$

$y \overset{u \lor x \lor y}{=} 1$

$z \overset{x \lor \overline{y} \lor z}{=} 1$

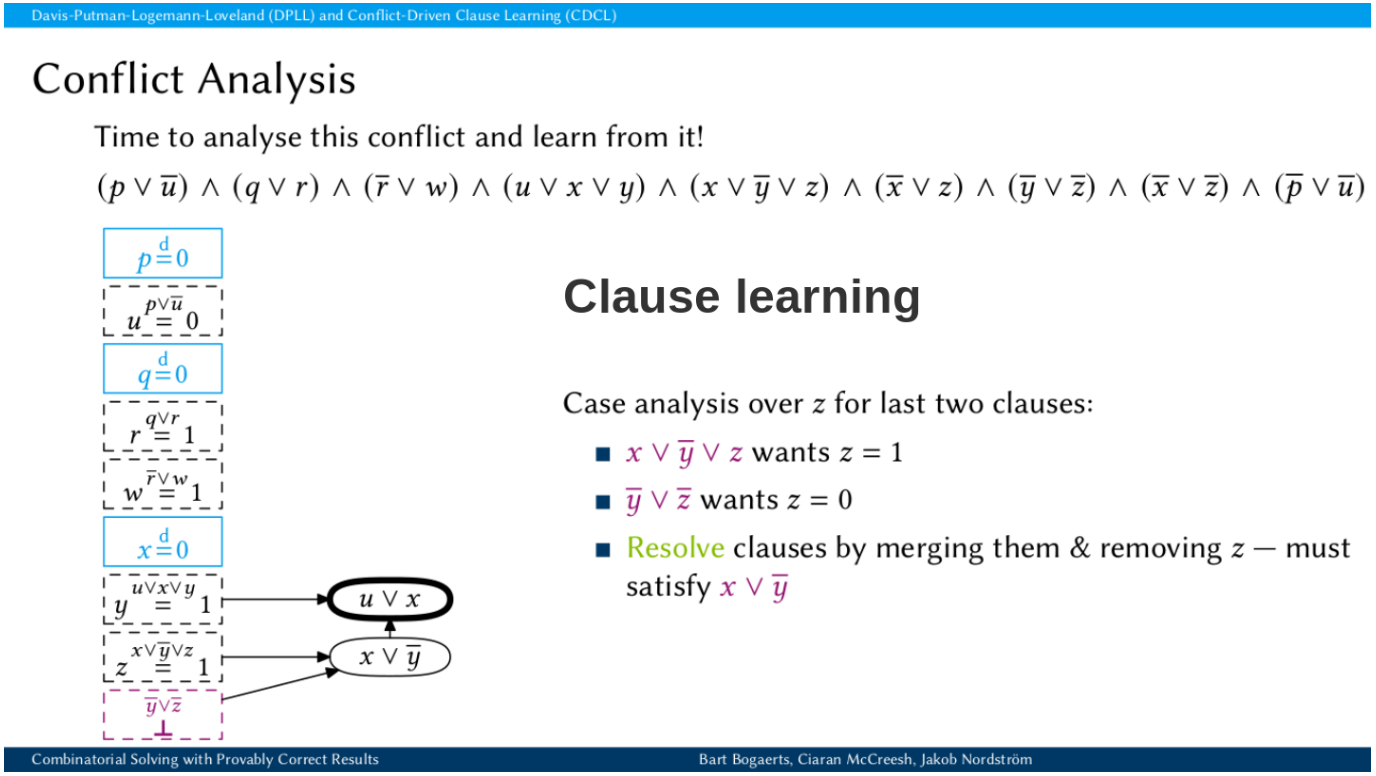
$\overset{\overline{y} \lor \overline{z}}{\bot}$

Decision

Unit propagation

Conflict detected

Deepdive: incremental CDCL solving with assumption variables 2/4



Conflict Analysis

Time to analyse this conflict and learn from it!

$$(p \vee \overline{u}) \wedge (q \vee r) \wedge (\overline{r} \vee w) \wedge (u \vee x \vee y) \wedge (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z}) \wedge (\overline{x} \vee \overline{z}) \wedge (\overline{p} \vee \overline{u})$$

$$\boxed{p \overset{\mathrm{d}}{=} 0}$$
$$u \overset{p \vee \overline{u}}{=} 0$$
$$\boxed{q \overset{\mathrm{d}}{=} 0}$$
$$r \overset{q \vee r}{=} 1$$
$$w \overset{\overline{r} \vee w}{=} 1$$
$$\boxed{x \overset{\mathrm{d}}{=} 0}$$
$$y \overset{u \vee x \vee y}{=} 1$$
$$z \overset{x \vee \overline{y} \vee z}{=} 1$$
$$\overset{\overline{y} \vee \overline{z}}{\perp}$$

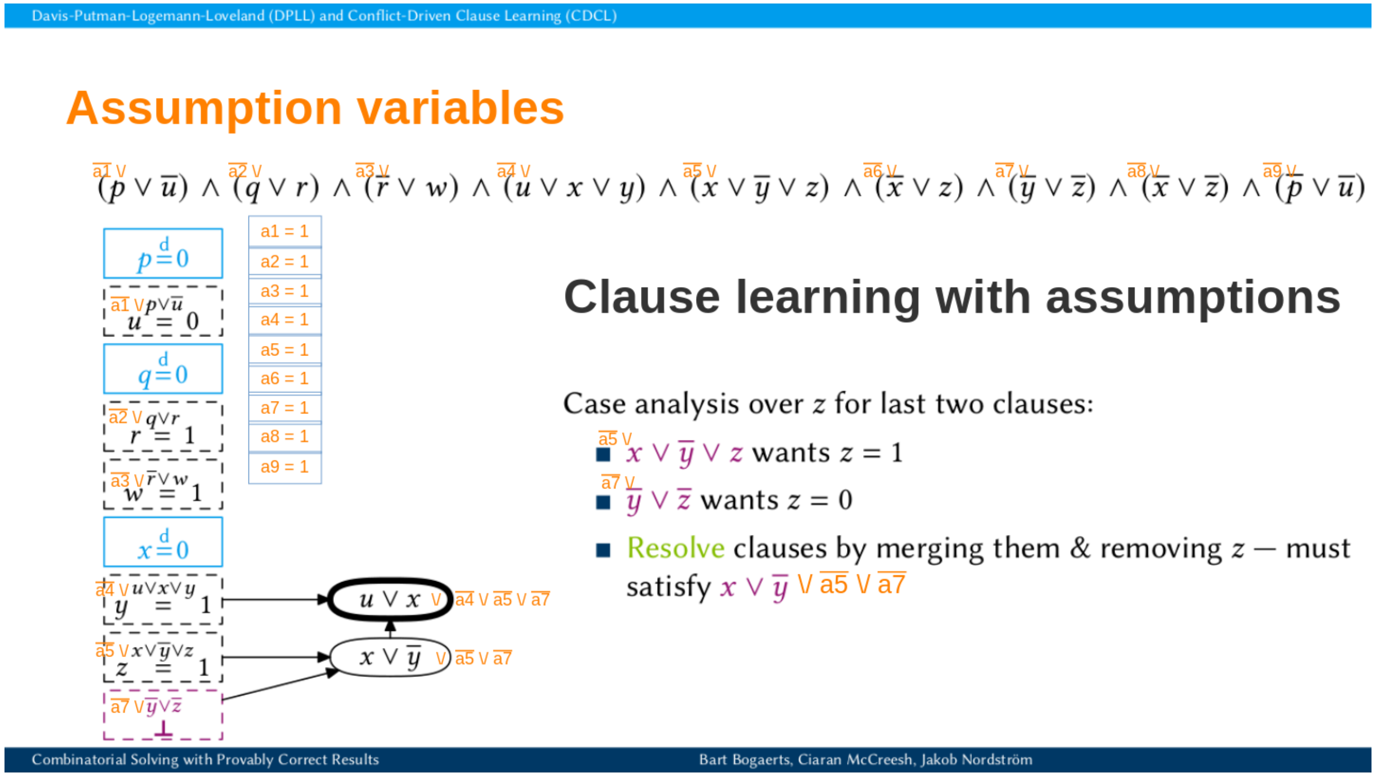
$$\begin{tikzcd}
y \overset{u \vee x \vee y}{=} 1 \arrow[r] & u \vee x \\
z \overset{x \vee \overline{y} \vee z}{=} 1 \arrow[r] & x \vee \overline{y} \arrow[u] \\
\overset{\overline{y} \vee \overline{z}}{\perp} \arrow[ru] &
\end{tikzcd}$$

Clause learning

Case analysis over z for last two clauses:

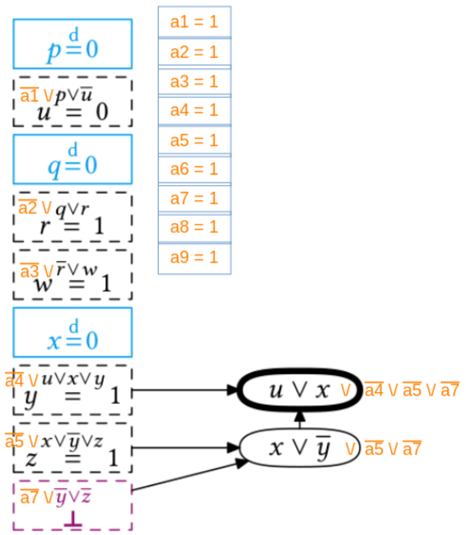
- $x \vee \overline{y} \vee z$ wants $z = 1$
- $\overline{y} \vee \overline{z}$ wants $z = 0$
- Resolve clauses by merging them & removing z — must satisfy $x \vee \overline{y}$

Deepdive: incremental CDCL solving with assumption variables 3/4



Assumption variables

$\overline{a1} \lor (p \lor \overline{u}) \land \overline{a2} \lor (q \lor r) \land \overline{a3} \lor (\overline{r} \lor w) \land \overline{a4} \lor (u \lor x \lor y) \land \overline{a5} \lor (x \lor \overline{y} \lor z) \land \overline{a6} \lor (\overline{x} \lor z) \land \overline{a7} \lor (\overline{y} \lor \overline{z}) \land \overline{a8} \lor (\overline{x} \lor \overline{z}) \land \overline{a9} \lor (\overline{p} \lor \overline{u})$



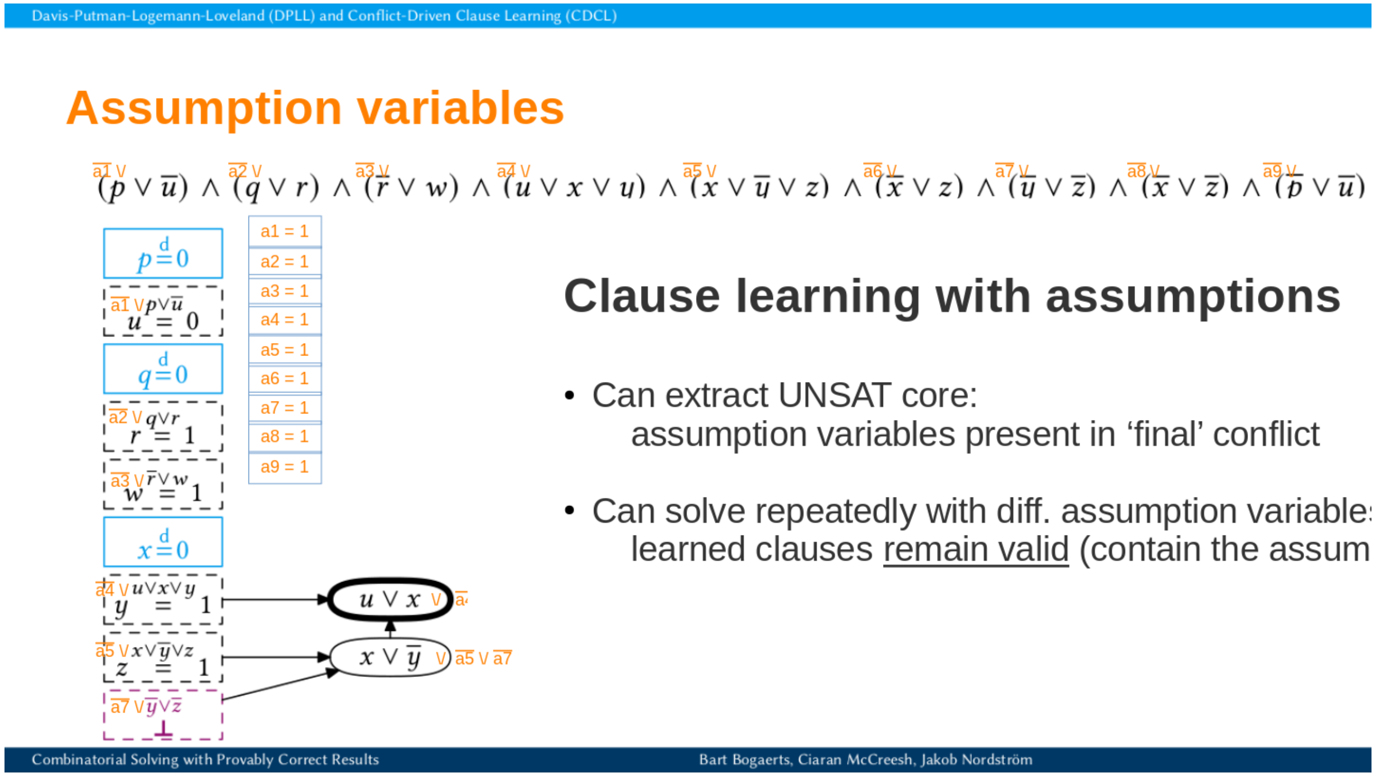
$p \overset{d}{=} 0$
$\overline{a1} \lor p \lor \overline{u}$
$u = 0$
$q \overset{d}{=} 0$
$\overline{a2} \lor q \lor r$
$r = 1$
$\overline{a3} \lor \overline{r} \lor w$
$w = 1$
$x \overset{d}{=} 0$
$\overline{a4} \lor u \lor x \lor y$
$y = 1$
$\overline{a5} \lor x \lor \overline{y} \lor z$
$z = 1$
$\overline{a7} \lor \overline{y} \lor \overline{z}$
$\perp$
a1 = 1
a2 = 1
a3 = 1
a4 = 1
a5 = 1
a6 = 1
a7 = 1
a8 = 1
a9 = 1
$u \lor x \lor \overline{a4} \lor \overline{a5} \lor \overline{a7}$
$x \lor \overline{y} \lor \overline{a5} \lor \overline{a7}$

Clause learning with assumptions

Case analysis over z for last two clauses:

- $\overline{a5} \lor x \lor \overline{y} \lor z$ wants $z = 1$
- $\overline{a7} \lor \overline{y} \lor \overline{z}$ wants $z = 0$
- Resolve clauses by merging them & removing z — must satisfy $x \lor \overline{y} \lor \overline{a5} \lor \overline{a7}$

Deepdive: incremental CDCL solving with assumption variables 4/4



Assumption variables

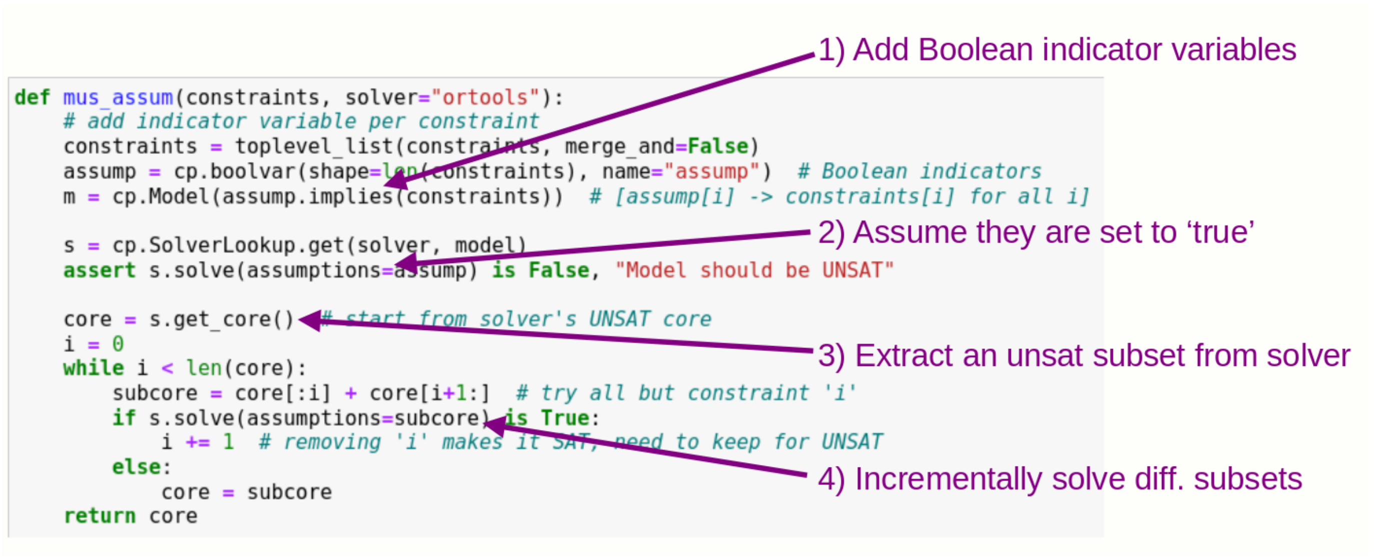
$(\overline{a1} \vee p \vee \overline{u}) \wedge (\overline{a2} \vee q \vee r) \wedge (\overline{a3} \vee \overline{r} \vee w) \wedge (\overline{a4} \vee u \vee x \vee y) \wedge (\overline{a5} \vee x \vee \overline{y} \vee z) \wedge (\overline{a6} \vee \overline{x} \vee z) \wedge (\overline{a7} \vee \overline{y} \vee \overline{z}) \wedge (\overline{a8} \vee \overline{x} \vee \overline{z}) \wedge (\overline{a9} \vee \overline{p} \vee \overline{u})$

$p \overset{d}{=} 0$
$\overline{a1} \vee p \vee \overline{u}$ $u = 0$
$q \overset{d}{=} 0$
$\overline{a2} \vee q \vee r$ $r = 1$
$\overline{a3} \vee \overline{r} \vee w$ $w = 1$
$x \overset{d}{=} 0$
$\overline{a4} \vee u \vee x \vee y$ $y = 1$
$\overline{a5} \vee x \vee \overline{y} \vee z$ $z = 1$
$\overline{a7} \vee \overline{y} \vee \overline{z}$ $\perp$
a1 = 1
a2 = 1
a3 = 1
a4 = 1
a5 = 1
a6 = 1
a7 = 1
a8 = 1
a9 = 1
$u \vee x \vee \overline{a4}$
$x \vee \overline{y} \vee \overline{a5} \vee \overline{a7}$

Clause learning with assumptions

- Can extract UNSAT core:
  assumption variables present in ‘final’ conflict
- Can solve repeatedly with diff. assumption variable
  learned clauses remain valid (contain the assum

How to compute a MUS, efficiently? (recap after deepdive)



```
def mus_assum(constraints, solver="ortools"):
    # add indicator variable per constraint
    constraints = toplevel_list(constraints, merge_and=False)
    assump = cp.boolvar(shape=len(constraints), name="assump")  # Boolean indicators
    m = cp.Model(assump.implies(constraints))  # [assump[i] -> constraints[i] for all i]

    s = cp.SolverLookup.get(solver, model)
    assert s.solve(assumptions=assump) is False, "Model should be UNSAT"

    core = s.get_core() # start from solver's UNSAT core
    i = 0
    while i < len(core):
        subcore = core[:i] + core[i+1:]  # try all but constraint 'i'
        if s.solve(assumptions=subcore) is True:
            i += 1  # removing 'i' makes it SAT, need to keep for UNSAT
        else:
            core = subcore
    return core
```

1) Add Boolean indicator variables
2) Assume they are set to 'true'
3) Extract an unsat subset from solver
4) Incrementally solve diff. subsets

How to compute a MUS, efficiently?

Assumption-based incremental solving only for Boolean SAT problems?

How to compute a MUS, efficiently?

Assumption-based incremental solving only for Boolean SAT problems?

No!

- CP solvers: *Lazy Clause Generation* (e.g. OrTools)
- Pseudo-Boolean solvers: *Conflict-Driven Cutting Plane Learning* (e.g. Exact)
- SMT solvers: *SAT Module Theories with CDCL* (e.g. Z3)
- MaxSAT solvers: *Core-guided solvers*

Deductive Explanations for UNSAT problems

A MUS is a deductive explanation of UNSAT:

these constraints minimally entail failure

Deductive Explanations for UNSAT problems

A MUS is a deductive explanation of UNSAT:

these constraints minimally entail failure

In [17]:
```
subset = cpmpy.tools.explain.mus(constraints)
print("Length of MUS:", len(subset))
for cons in subset: print("-", cons)
```

```
Length of MUS: 11
- Shift D on Sat 1 must be covered by 5 nurses out of 8
- Robert can work at most 5 days before having a day off
- Kevin should work at most 1 weekends
- Katherine has a day off on Sat 1
- Richard has a day off on Sat 1
- Robert requests to work shift D on Mon 1
- Robert requests to work shift D on Tue 1
- Robert requests to work shift D on Wed 1
- Robert requests to work shift D on Thu 1
- Robert requests to work shift D on Fri 1
- Kevin requests to work shift D on Sun 2
```

In [18]:
```
visualize_constraints(subset, nurse_view, factory)
```

Out[18]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine													
Robert													
Jonathan													
William													
Richard													
Kristen													
Kevin													
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

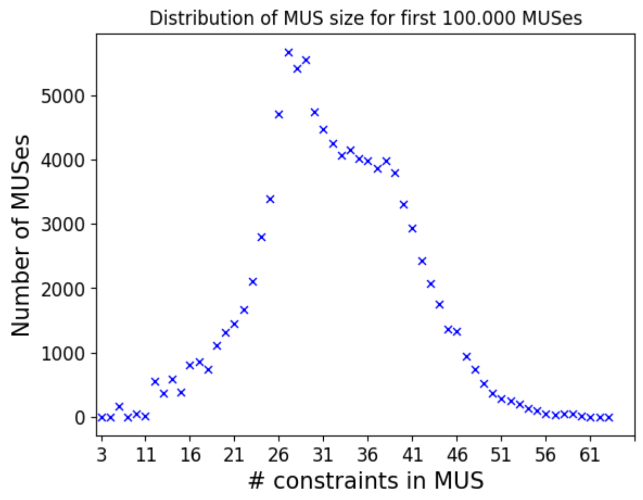
Many MUS'es may exist...

Liffiton, M.H., & Malik, A. (2013). Enumerating infeasibility: Finding multiple MUSes quickly. In Proceedings of the 10th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2013) (pp. 160–175)

In [19]:
```python
# MARCO MUS/MSS enumeration
from explanations.marco_mcs_mus import do_marco
solver = "ortools"  # default solver
if "exact" in cp.SolverLookup.solvernames(): solver = "exact"  # fast fo

t0 = time.time()
cnt = 0
for (kind, sset) in do_marco(model, solver=solver):
    if kind == "MUS":
        print("M", end="")
        cnt += 1
    else: print(".", end="")  # MSS

    if time.time() - t0 > 15:  break  # for this presentation: break aft
print(f"\nFound {cnt} MUSes in", time.time() - t0)
```

```
MMMMMMMMMMMMM.MMMMMMM.MMMM.MMMMMMMMMM.MMMMMMMMMMM..MMMMMMMM.MMMM
MMMMMMMMMMMMM.MMMMMMMMMM.MMMM..MMMMMMMMMMMM..M.M.M..MMM..MM.
M.....MMMMMMMM.MMMM..MMMMMMMM..MMMMMMM..MMMMM....MMMMMMM.MMMMMMM
MM..MMMMMMMMM.MM.M.MMM.MMM..MMMMMMMMM..M..MMMMMMMMM.M...M.MMMM..
MMMMMM
Found 202 MUSes in 15.035604476928711
```

Many MUS'es may exist...



This problem has just 168 constraints, yet 100.000+ MUSes exist...

Which one to show?

Can we influence which MUS is found?

Influencing which MUS is found?

QuickXPlain algorithm *(Junker, 2004)*. Widely used, in model-based diagnosis, recommender systems, verification, and more.

Divide-and-conquer given a lexicographic *preference* order over the constraints:

Influencing which MUS is found?

QuickXPlain algorithm *(Junker, 2004)*. Widely used, in model-based diagnosis, recommender systems, verification, and more.

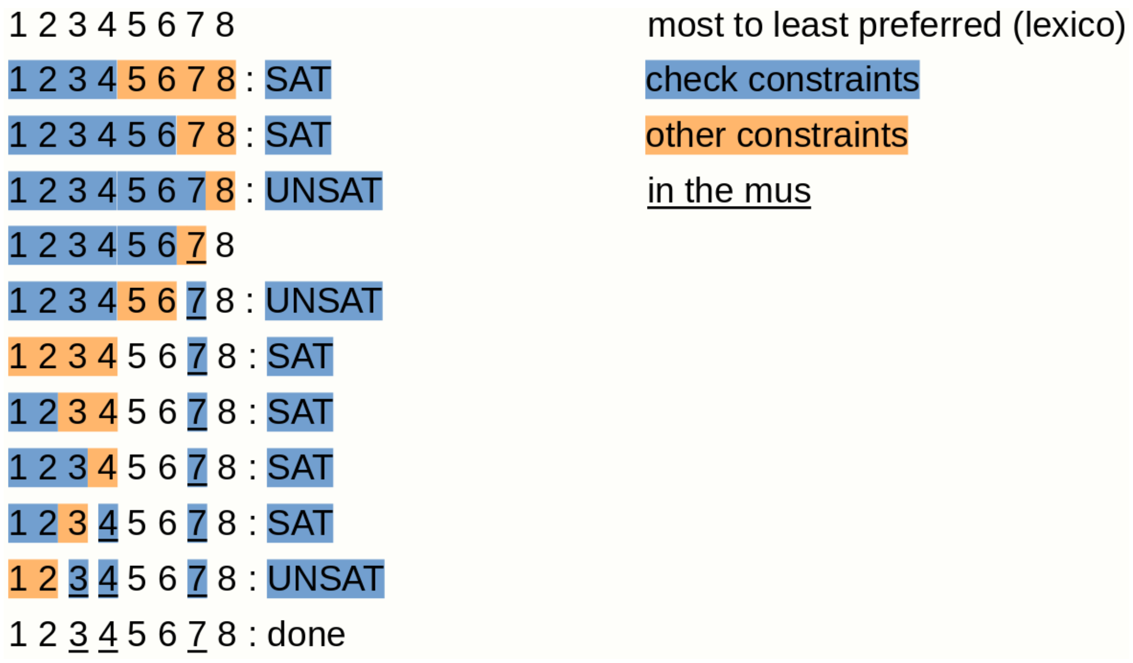
Divide-and-conquer given a lexicographic *preference* order over the constraints:

In [20]:
```
# the order of 'soft' matters! lexicographic preference for the first or
def quickxplain(soft, hard=[], solver="ortools"):
    model, soft, assump = make_assump_model(soft, hard)
    s = cp.SolverLookup.get(solver, model)
    assert s.solve(assumptions=assump) is False, "The model should be UN

    # the recursive call
    def do_recursion(tocheck, other, delta):
        if len(delta) != 0 and s.solve(assumptions=tocheck) is False:
            # conflict is in hard constraints, no need to recurse
            return []

        if len(other) == 1:
            # conflict is not in 'tocheck' constraints, but only 1 'othe
            return list(other)  # base case of recursion

        split = len(other) // 2  # determine split point
        more_preferred, less_preferred = other[:split], other[split:]  #

        # treat more preferred part as hard and find extra constants fro
        delta2 = do_recursion(tocheck + more_preferred, less_preferred,
        # find which preferred constraints exactly
```

```
            delta1 = do_recursion(tocheck + delta2, more_preferred, delta2)
            return delta1 + delta2

    core = do_recursion([], list(assump), [])
    return [c for c,var in zip(soft,assump) if var in core]
```

Influencing which MUS is found?

QuickXPlain: Divide-and-conquer given a lexicographic *preference* order over the constraints:



1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8 : SAT

1 2 3 4 5 6 7 8 : SAT

1 2 3 4 5 6 7 8 : UNSAT

1 2 3 4 5 6 $\underline{7}$ 8

1 2 3 4 5 6 $\underline{7}$ 8 : UNSAT

1 2 3 4 5 6 $\underline{7}$ 8 : SAT

1 2 3 4 5 6 $\underline{7}$ 8 : SAT

1 2 3 4 5 6 $\underline{7}$ 8 : SAT

1 2 3 $\underline{4}$ 5 6 $\underline{7}$ 8 : SAT

1 2 $\underline{3}$ $\underline{4}$ 5 6 $\underline{7}$ 8 : UNSAT

1 2 $\underline{3}$ $\underline{4}$ 5 6 $\underline{7}$ 8 : done

Influencing which MUS is found?

QuickXPlain algorithm *(Junker, 2004)*. Widely used, in model-based diagnosis, recommender systems, verification, and more.

Divide-and-conquer given a lexicographic order over the constraints

```
In [21]: t0 = time.time()
         subset = cpmpy.tools.explain.quickxplain(sorted(model.constraints, key=l
         print("ordering '-len': Length of MUS:", len(subset))
         print(f"(in {time.time()-t0} seconds)")

         t0 = time.time()
         subset = cpmpy.tools.explain.quickxplain(sorted(model.constraints, key=l
         print("ordering 'len': Length of MUS:", len(subset))
         print(f"(in {time.time()-t0} seconds)")
```

```
ordering '-len': Length of MUS: 18
(in 2.670808792114258 seconds)
ordering 'len': Length of MUS: 3
(in 2.420356273651123 seconds)
```

Influencing which MUS is found?

QuickXPlain algorithm *(Junker, 2004)*. Widely used, in model-based diagnosis, recommender systems, verification, and more.

Divide-and-conquer given a lexicographic order over the constraints

In [21]:
```
t0 = time.time()
subset = cpmpy.tools.explain.quickxplain(sorted(model.constraints, key=l
print("ordering '-len': Length of MUS:", len(subset))
print(f"(in {time.time()-t0} seconds)")

t0 = time.time()
subset = cpmpy.tools.explain.quickxplain(sorted(model.constraints, key=l
print("ordering 'len': Length of MUS:", len(subset))
print(f"(in {time.time()-t0} seconds)")
```

```
ordering '-len': Length of MUS: 18
(in 2.670808792114258 seconds)
ordering 'len': Length of MUS: 3
(in 2.420356273651123 seconds)
```

In [22]:
```
t0 = time.time()
subset = cpmpy.tools.explain.quickxplain(sorted(model.constraints, key=l
print("ordering 'len': Length of MUS:", len(subset))
print(f"(in {time.time()-t0} seconds)")
```

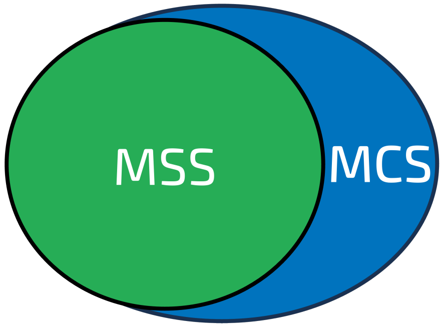
```
ordering 'len': Length of MUS: 3
(in 2.810506582260132 seconds)
```

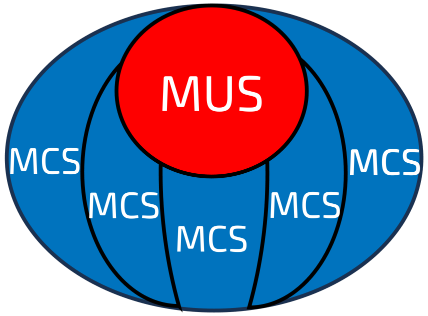
Optimising which MUS is found?

Give every constraint a weight: OUS: Optimal Unsatisfiable Subsets *(Gamba, Bogaerts, Guns, 2021)*.

Some key properties:

1. If a subset is SAT, can *grow* it to a Maximal Satisfiable Subset (MSS)
2. The complement of a MSS is a Minimum Correction Subset (MCS)
3. Theorem: A MUS is a hitting set of the MCSes

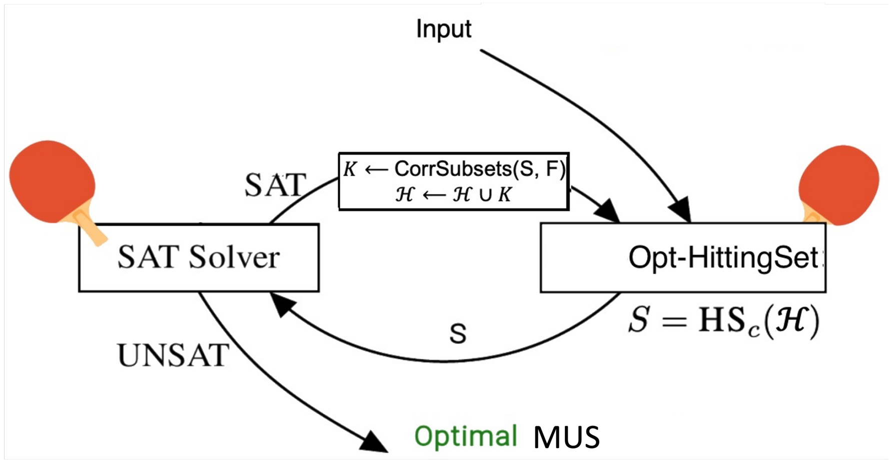




Optimising which MUS is found?

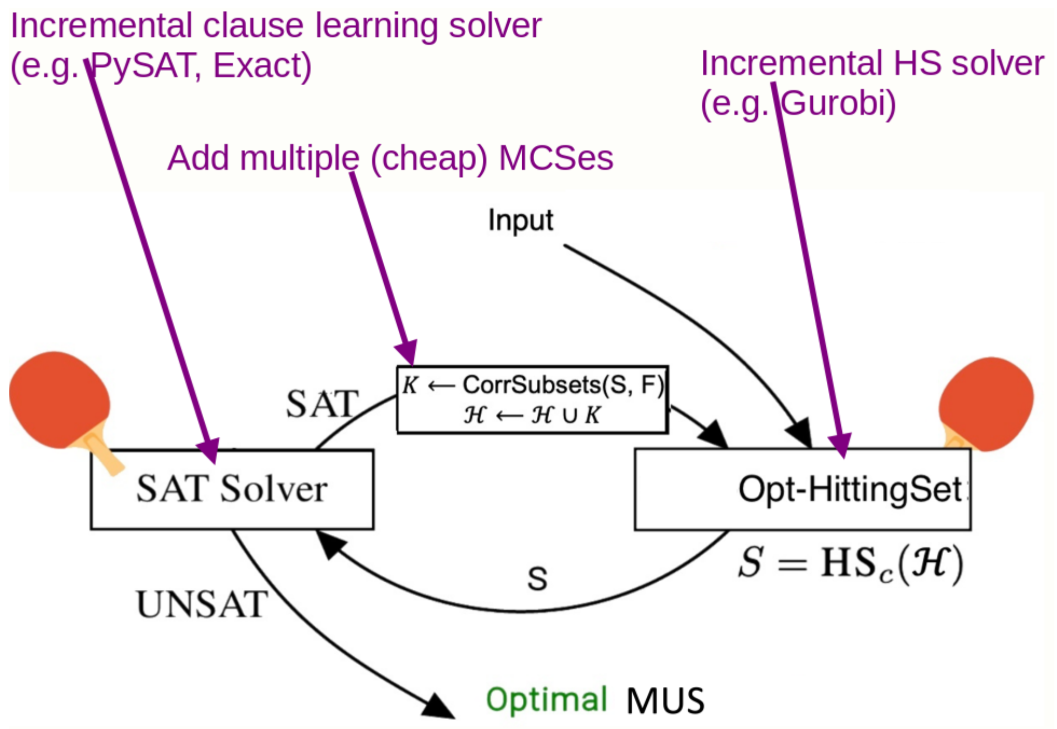
OUS: Optimal Unsatisfiable Subsets *(Gamba, Bogaerts, Guns, 2021)*. Every constraints has a weight.

1. Initialize sets-to-hit $\mathcal{H}$ (e.g. insert set of all constraints)
2. Find *optimal* hitting set S
3. Check if SAT: grow and take complement = MCS K, add to sets-to-hit $\mathcal{H}$
4. Repeat until UNSAT: optimal unsatisfiable subset S found


Input
SAT
$K \leftarrow$ CorrSubsets(S, F)
$\mathcal{H} \leftarrow \mathcal{H} \cup K$
SAT Solver
Opt-HittingSet
$S = \mathbf{HS}_c(\mathcal{H})$
S
UNSAT
Optimal MUS

Efficiently optimising which MUS is found?

OUS: Optimal Unsatisfiable Subsets *(Gamba, Bogaerts, Guns, 2021)*. Every constraints has a weight.


Incremental clause learning solver (e.g. PySAT, Exact)
Incremental HS solver (e.g. Gurobi)
Add multiple (cheap) MCSes
Input
SAT
$K \leftarrow$ CorrSubsets(S, F)
$\mathcal{H} \leftarrow \mathcal{H} \cup K$
SAT Solver
Opt-HittingSet
$S = \mathrm{HS}_c(\mathcal{H})$
S
UNSAT
Optimal MUS

Optimising which MUS is found?

OUS: Optimal Unsatisfiable Subsets *(Gamba, Bogaerts, Guns, 2021)*. Every constraints has a weight.

In [24]:
```
from explanations.subset import omus  # not (yet) part of CPMpy

smallest_subset = omus(model.constraints, weights=1, solver="exact", hs_

print("Length of OUS:", len(smallest_subset))
for cons in smallest_subset:
    print("-", cons)
```

```
Length of OUS: 3
- Robert has a day off on Tue 2
- Richard requests to not work shift D on Tue 2
- Shift D on Tue 2 must be covered by 7 nurses out of 8
```

In [25]:
```
visualize_constraints(smallest_subset, nurse_view, factory)
```

Out[25]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine													
Robert													
Jonathan													
William													
Richard													
Kristen													
Kevin													
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets
    - efficient MUSes
    - preferred MUSes
- SAT: explaining logical consequences ⏪
- OPT: explaining that no better solution exists

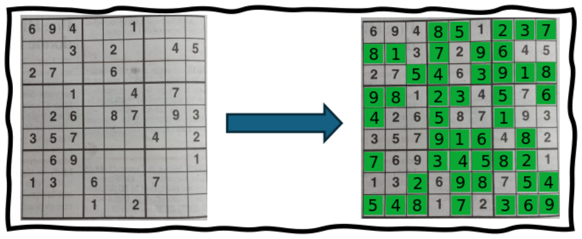
Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function

Deductive Explanations for SAT problems

How to explain satisfiability of a constraint satisfaction problem (CSP) in a human-understandable way?

Explain the <u>maximal consequence</u> of a CSP



I_0 I_{end}

$\boxed{C}$ ***constraints***: e.g. all different values on a row, block, and column

I_0 initial **(partial) interpretation** (facts):

- ∅
- `cells[1,1]=6,cells[1,2]=9,…,cells[8,7]=7`

I_{end} **maximal consequence:** *precision-maximal partial interpretation*

$$\boxed{C} \wedge I_0 \models I_{end}$$

Deductive Explanations for SAT problems

Explaining logical consequences

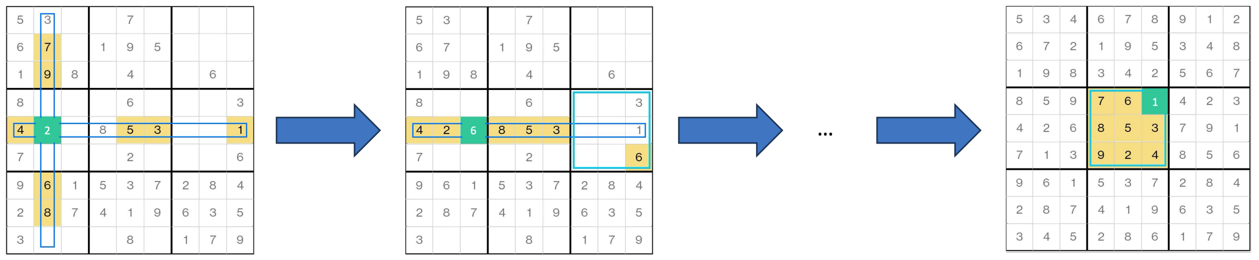
Logical consequence: a variable assignment entailed by the constraints and the current partial assignment

Maximal consequence: precision- maximal partial assignment

- Maximal consequence = intersection of all possible solutions
- If solution is unique, maximal consequence = unique solution

Deductive Explanations for SAT problems

> *Bogaerts, Bart, Emilio Gamba, and Tias Guns. "A framework for step-wise explaining how to solve constraint satisfaction problems." Artificial Intelligence 300 (2021): 103550.*



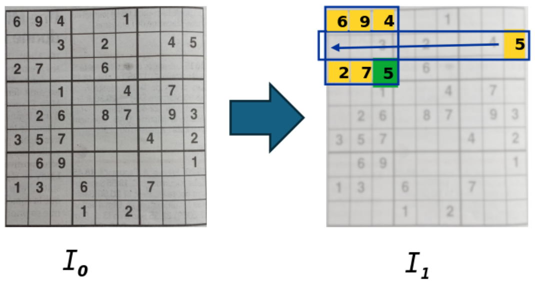
$cell_{2,2} = 7 \wedge cell_{3,2} = 9 \ldots \wedge$
$alldiff(row_5) \wedge alldiff(col_2) \Rightarrow cell_{5,2} = 2$

$cell_{5,1} = 4 \wedge cell_{5,2} = 2 \ldots \wedge$
$alldiff(row_5) \wedge alldiff(block_6) \Rightarrow cell_{5,3} = 6$

$cell_{4,4} = 7 \wedge cell_{4,5} = 6 \ldots \wedge$
$alldiff(block_5) \Rightarrow cell_{5,3} = 6$

Deductive Explanations for SAT problems

> *Bogaerts, Bart, Emilio Gamba, and Tias Guns. "A framework for step-wise explaining how to solve constraint satisfaction problems." Artificial Intelligence 300 (2021): 103550.*



An EXPLANATION (E_i, S_i, N_i) of an inference step ***explains***:

$$E_i \wedge \boxed{S_i} \models N_i$$

E_i — $\mathbf{E_i} \subseteq I_i$ — *The explaining facts are a subset of what was previously derived*

```
E₀ = {cells[1,1] = 6, cells[1,2] = 9, cells[1,3] = 4,
      cells[3,1] = 2, cells[3,2] = 7, cells[2,9] = 5}
```

$\boxed{S_i}$ — $\mathbf{S_i} \subseteq C$ — *A subset of the problem constraints*

```
S₀ = {alldiff(cells[1:3, 1:3]), alldiff(cells[2, :])}
```

N_i — $I_{i+1} \setminus I_i$ — *All newly derived information entailed by this explanation*

```
N₀ = {cells[3,3] = 5}
```

Deductive Explanations for SAT problems

We want each explanation step to be as simple as possible.

An EXPLANATION (E_i, S_i, N_i) of an inference step ***explains***:

$$\boxed{E_i} \wedge \boxed{S_i} \models \boxed{N_i}$$

$\boxed{E_i}$	$\mathbf{E_i} \subseteq \boldsymbol{I_i}$	*The explaining facts are a subset of what was previously derived*
$\boxed{S_i}$	$\mathbf{S_i} \subseteq \boldsymbol{C}$	*A subset of the problem constraints*
$\boxed{N_i}$	$\boldsymbol{I_{i+1}} \setminus \boldsymbol{I_i}$	*All newly derived information entailed by this explanation*

How? $\mathcal{MUS}(\boxed{I} \wedge \boxed{C} \wedge \boxed{-n})$

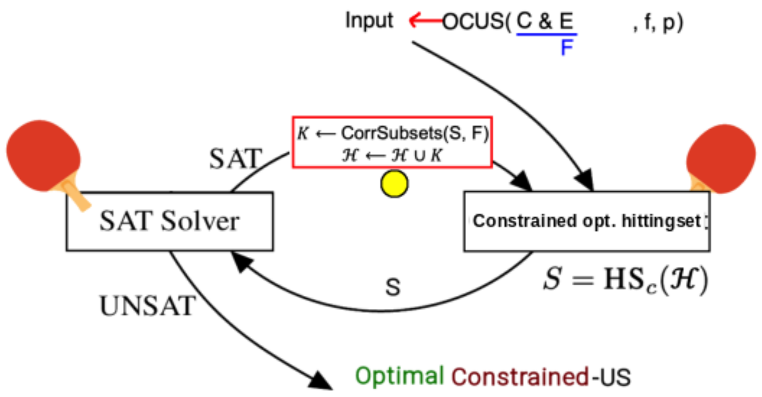
(we actually use OUS because we want the *smallest* not just a minimal one, and then we can put smaller weights on facts and larger weights on constraints)

Efficiently step-wise explanation of the maximal consequence?

Compute the OUS over all assignments in the maximal consequence at once, efficiently:

OCUS Optimal *Constrained* Unsatisfiable Subsets *(Gamba, Bogaerts, Guns, 2021)*.

- *meta-constraint p:* use exactly 1 element of the maximal consequence



Input
OCUS(C & E / F , f, p)
$K \leftarrow$ CorrSubsets(S, F)
$\mathcal{H} \leftarrow \mathcal{H} \cup K$
SAT
SAT Solver
Constrained opt. hittingset
S
$S = \mathrm{HS}_c(\mathcal{H})$
UNSAT
Optimal Constrained-US

(not discussed in more detail)

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets
    - efficient MUSes
    - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists ⏪

Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function

Deductive Explanations for OPT problems

Can we explain *why* an optimal solution is optimal, e.g. why there does not exist a better solution?

A *proof of optimality* proves that no better solution exists, but:

- An increasing number of solvers support *proof logging* (SAT, but also CP: Glasgow Constraint Solver)
- These proofs are built for *computer* verification (up to gigabytes of log), not to communicate to users
- These proofs can use learned clauses, auxiliary variables and anything available to the solver

Deductive Explanations for OPT problems

Can we explain *why* an optimal solution is optimal, e.g. why there does not exist a better solution?

Let be the constraints, the objective function and the optimal objective value.

- **because** of the constraints
- Hence is unsatisfiable...
- Hence is a deductive explanation for optimality!

Deductive Explanations for OPT problems

Can we explain *why* an optimal solution is optimal, e.g. why there does not exist a better solution?

Let C be the constraints, $f(x)$ the objective function and o the optimal objective value.

- $o = min_{x \in C} f(x)$ **because** of the constraints C
- Hence $C \ \wedge \ (f(x) < o)$ is unsatisfiable...
- Hence $\mathrm{MUS}(C \ \wedge \ (f(x) < o))$ is a deductive explanation for optimality!

But its typically very big (up to all constraints)...

can we provide a **step-wise explanation** of the unsatisfiability?

Deductive Explanations for OPT problems

Can we explain *why* an optimal solution is optimal, e.g. a step-wise explanation of why there does not exist a better solution?

Yes!

> *Ignace Bleukx, Jo Devriendt, Emilio Gamba, Bart Bogaerts, Tias Guns. Simplifying Step-wise Explanation Sequences. 29th International Conference on Principles and Practice of Constraint Programming (CP23), 2023.*

Challenges

- How to find interpretable sequences?
    - *I.e., with few and small steps?*
- How to deal with redundancy in the sequence?
    - *I.e., how to decide what information is relevant to derive?*
- How to make the algorithm incremental?
    - *I.e., how to find good sequences fast?*

Deductive Explanations for <u>OPT</u> problems

Example in this tutorial: step-wise explanation of a large MUS
(can also construct from scratch to step-wise explain optimality, see paper)

In [27]:
```python
# any MUS
subset = cpmpy.tools.explain.mus(model.constraints)
visualize_constraints(subset, nurse_view, factory)
```

Out[27]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine													
Robert													
Jonathan													
William													
Richard													
Kristen													
Kevin													
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [28]: from explanations.stepwise import find_sequence

         seq = find_sequence(subset)
```

```
Found sequence of length 11
Filtered sequence to length 11
```

```
In [29]: nurse_view.clear()
         visualize_step(seq[0], nurse_view, factory)
```

```
Propagating constraint: Katherine has a day off on Sat 1
```

Out[29]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert													
Jonathan													
William													
Richard													
Kristen													
Kevin													
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [30]: visualize_step(seq[1], nurse_view, factory)

Propagating constraint: Richard has a day off on Sat 1

Out[30]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert													
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	0/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [31]: visualize_step(seq[2], nurse_view, factory)
```

```
Propagating constraint: Robert requests to work shift D on Mon 1
```

```
Out[31]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D												
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	0/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [32]: visualize_step(seq[3], nurse_view, factory)

Propagating constraint: Robert requests to work shift D on Tue 1

Out[32]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D											
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	1/7	0/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [33]: visualize_step(seq[4], nurse_view, factory)
```

```
Propagating constraint: Robert requests to work shift D on Wed 1
```

Out[33]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D										
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	1/7	1/6	0/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [34]: visualize_step(seq[5], nurse_view, factory)

         Propagating constraint: Robert requests to work shift D on Thu 1
```

Out[34]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D	D									
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	1/7	1/6	1/4	0/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [35]: visualize_step(seq[6], nurse_view, factory)
```

```
Propagating constraint: Robert requests to work shift D on Fri 1
```

```
Out[35]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D	D	D								
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	1/7	1/6	1/4	1/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [36]: visualize_step(seq[7], nurse_view, factory)
```

```
Propagating constraint: Robert can work at most 5 days before ha
ving a day off
```

Out[36]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D	D	D	F							
Jonathan													
William													
Richard						F							
Kristen													
Kevin													
Cover D	1/5	1/7	1/6	1/4	1/5	0/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [37]: visualize_step(seq[8], nurse_view, factory)

         Propagating constraint: Shift D on Sat 1 must be covered by 5 nu
         rses out of 8

Out[37]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D	D	D	F							
Jonathan													
William													
Richard						F							
Kristen													
Kevin						D							
Cover D	1/5	1/7	1/6	1/4	1/5	1/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [38]: visualize_step(seq[9], nurse_view, factory)
```

```
Propagating constraint: Kevin should work at most 1 weekends
```

```
Out[38]:
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan													
Katherine						F							
Robert	D	D	D	D	D	F							
Jonathan													
William													
Richard						F							
Kristen													
Kevin						D							
Cover D	1/5	1/7	1/6	1/4	1/5	1/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

```
In [39]: visualize_step(seq[10], nurse_view, factory)
```

```
Propagating constraint: Kevin requests to work shift D on Sun 2
```

```
Out[39]:
```

	Week 1							Week 2					
	Mon	**Tue**	**Wed**	**Thu**	**Fri**	**Sat**	**Sun**	**Mon**	**Tue**	**Wed**	**Thu**	**Fri**	**Sat**
name													
Megan													
Katherine						F							
Robert	D	D	D	D	D	F							
Jonathan													
William													
Richard						F							
Kristen													
Kevin						D							
Cover D	1/5	1/7	1/6	1/4	1/5	1/5	0/5	0/6	0/7	0/4	0/2	0/5	0/6

Outline of the talk

Part 1: Deductive explanations (What causes X?) ✅

- UNSAT: minimal unsatisfiable subsets
    - efficient MUSes
    - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists

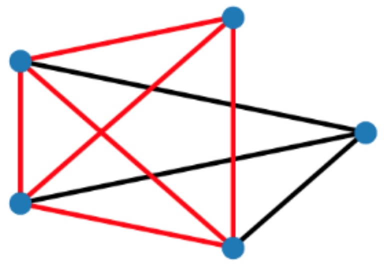
Part 2: Counterfactual explanation (What if Y instead of X?) ⏪

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function

Explainable Constraint Programming (XCP)

Recap, "**Why X?**" (with X a solution or UNSAT)

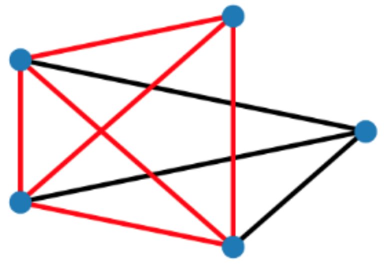
- **Deductive explanation:**
  - *What causes X?*
  - answer: a minimal inference set



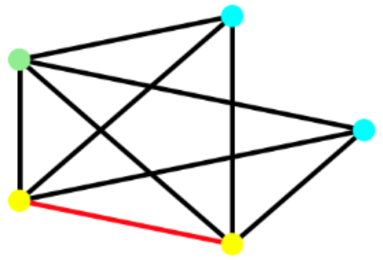
Explainable Constraint Programming (XCP)

Recap, "**Why X?**" (with X a solution or UNSAT)

- **Deductive explanation:**
  - *What causes X?*
  - answer: a minimal inference set

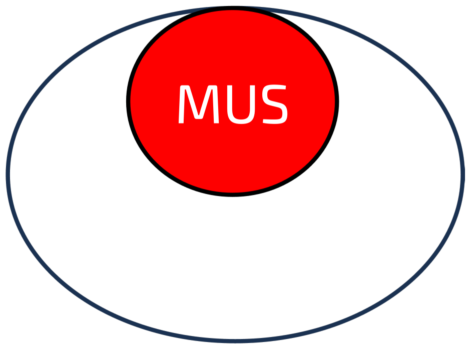


- **Counterfactual explanation**:
  - *What if I want Y instead of X?*
  - answer: a constraint relaxation + new solution

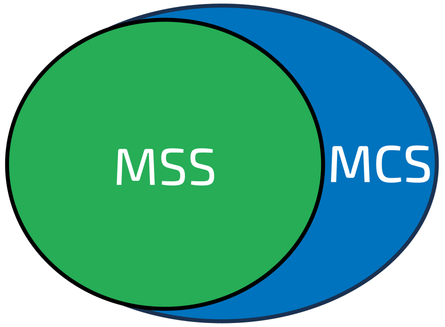


Explanations for UNSAT problems:

MUS: one conflict



MSS: a relaxation



Counterfactual Explanations for UNSAT problems

Computing a *Maximal Satisfiable Subset*?

We can do better… computing a Maxi**mum** satisfiable subset is the textbook MaxSAT/MaxCSP problem!

Can add Boolean indicator variable to every constraint (like in assumption-based solving), and maximize the sum of indicators…

Counterfactual Explanations for UNSAT problems

Computing a *Maximal Satisfiable Subset*?

We can do better... computing a Maxi**mum** satisfiable subset is the textbook MaxSAT/MaxCSP problem!

Can add Boolean indicator variable to every constraint (like in assumption-based solving), and maximize the sum of indicators...

```
In [40]:
# add indicator variable per expression
constraints = toplevel_list(model.constraints, merge_and=False)

ind = cp.boolvar(shape=len(constraints), name="ind")  # Boolean indicato
ind_model = cp.Model(ind.implies(constraints))
ind_model.maximize(sum(ind))

ind_model.solve()
print(ind_model.status(), "\n")

print("MSS: size =", sum(ind.value()),"constraints")
print("MCS:")
for a,c in zip(ind, constraints):
    if not a.value(): print("-",c)
```

```
ExitStatus.OPTIMAL (0.044801015 seconds)

MSS: size = 164 constraints
MCS:
- Robert has a day off on Tue 2
- Richard requests to not work shift D on Tue 2
- Shift D on Sat 1 must be covered by 5 nurses out of 8
- Shift D on Sun 1 must be covered by 5 nurses out of 8
```

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets
    - efficient MUSes
    - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists

Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions ⏪
- SAT: checking a foil
- OPT: correcting the objective function

Counterfactual Explanations for UNSAT problems

An MSS is a **relaxation** of the original problem.

- but *deleting* constraints is a very intrusive action!
- e.g. no requirement at all on number of nurses on Sat 1 and Sun 1?

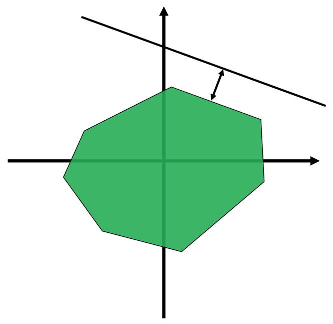
In [41]:
```
visualize(nurse_view.value(), factory, highlight_cover=True)
```

Out[41]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan	F	D	D	D	D	F	F	D	D	D	F	F	D
Katherine	D	D	D	D	D	F	F	D	D	F	F	D	D
Robert	D	D	D	D	D	F	F	D	D	F	F	F	F
Jonathan	D	D	F	F	D	D	F	F	D	D	D	D	F
William	F	D	D	F	F	F	F	D	D	F	F	D	D
Richard	D	D	D	F	F	F	F	D	D	F	F	D	D
Kristen	F	F	D	D	D	F	F	D	D	D	F	F	D
Kevin	D	D	F	F	F	F	F	F	F	D	D	D	D
Cover D	5/5	7/7	6/6	4/4	5/5	1/5	0/5	6/6	7/7	4/4	2/2	5/5	6/6

Counterfactual Explanations for UNSAT problems

Defining a relaxation space: *corrective actions* on the constraints



- Boolean constraints can only be turned on/off
- Numeric comparison constraints can be **violated** to some extend
  - Introduce slack for each numerical comparison
  - Slack indicates how much a constraint may be violated

    = fine grained penalty of solution!
- Minimize sum of slack and indicator values

Still a standard optimisation problem, just finer-grained correction modelling

> *Senthooran I, Klapperstueck M, Belov G, Czauderna T, Leo K, Wallace M, Wybrow M, Garcia de la Banda M. Human-centred feasibility restoration in practice. Constraints. 2023 Jul 20:1-41.*

Counterfactual Explanations for UNSAT problems

Detailed example: allowing 'over' and 'under' assigning a shift, with the Count global constraint.

```
In [42]: # slack variables can only be positive here (separate over and under rel
         slack_under = cp.intvar(0, len(data.staff), shape=data.horizon, name="sl
         slack_over = cp.intvar(0, len(data.staff), shape=data.horizon, name="sla

         for _, cover in factory.data.cover.iterrows():
             # read the data
             day = cover["# Day"]
             shift = factory.shift_name_to_idx[cover["ShiftID"]]

             nb_nurses = cp.Count(nurse_view[:, day], shift)
             # deviation of `nb_nurses` from `requirement`
             expr = (nb_nurses == cover["Requirement"] - slack_under[day] + slack
```

Counterfactual Explanations for UNSAT problems

Defining a relaxation space: *corrective actions* on the constraints.

```
In [43]: slack_model, slack_nurse_view, slack_under, slack_over = factory.get_sla
         slack_model.minimize(10*cp.max(slack_under) + cp.sum(slack_under) + 0.1*
         slack_model.solve()
         print(slack_model.status())
```

```
ExitStatus.OPTIMAL (0.031282838 seconds)
```

Counterfactual Explanations for UNSAT problems

Defining a relaxation space: *corrective actions* on the constraints.

```
In [43]:
slack_model, slack_nurse_view, slack_under, slack_over = factory.get_sla
slack_model.minimize(10*cp.max(slack_under) + cp.sum(slack_under) + 0.1*
slack_model.solve()
print(slack_model.status())
```

```
ExitStatus.OPTIMAL (0.031282838 seconds)
```

```
In [44]:
style = visualize(slack_nurse_view.value(), factory, highlight_cover=Tru
style.data.loc["Slack under"] = list(slack_under.value()) + [" "]
style.data.loc["Slack over"] = list(slack_over.value()) + [" "]
display(style)
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan	F	D	D	D	D	F	F	D	D	D	F	F	D
Katherine	D	D	D	D	D	F	F	D	D	F	F	D	D
Robert	D	D	D	D	D	F	F	F	F	D	D	D	F
Jonathan	D	D	F	F	F	D	D	D	D	D	F	F	F
William	D	D	D	F	F	D	D	D	D	F	F	F	F
Richard	D	D	D	D	D	F	F	F	F	F	F	D	D
Kristen	F	F	D	D	D	F	F	D	D	F	F	D	D
Kevin	D	D	F	F	F	F	F	F	F	D	D	D	D
Cover D	6/5	7/7	6/6	5/4	5/5	2/5	2/5	5/6	5/7	4/4	2/2	5/5	5/6
Slack under	0	0	0	0	0	3	3	1	2	0	0	0	1
Slack over	1	0	0	1	0	0	0	0	0	0	0	0	0

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets
  - efficient MUSes
  - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists

Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil ⏪
- OPT: correcting the objective function

Counterfactual Explanations for SAT problems

The problem is SATisfiable, and the solver returned a solution.

The user asks: "What if Y instead of X?"

Y is a foil: a partial assignment or constraint that is counter-factual, different from the returned solution.

Counterfactual Explanations for SAT problems

The problem is SATisfiable, and the solver returned a solution.

The user asks: "What if Y instead of X?"

Y is a foil: a partial assignment or constraint that is counter-factual, different from the returned solution.

Need to check $C + Y$, with C the set of constraints and Y the foil

- If $C + Y$ is also SAT: show this solution
- If $C + Y$ is UNSAT: can show a deductive or counterfactual explanation of why the foil leads to UNSAT

Counterfactual Explanations for SAT problems

Example where the user asks: "What if Y instead of X?"

In [45]:

```
assert nurse_view[4,5].value()  # William currently scheduled to work or
v = slack_model.objective_value()

# what if William would not work on the first Saturday?
mmodel = slack_model.copy()
mmodel += (nurse_view[4,5] == 0)

assert mmodel.solve()
print("Total penalty: ", mmodel.objective_value(), "versus", v, "before.
style = visualize(slack_nurse_view.value(), factory, highlight_cover=Tru
style.data.loc["Slack under"] = list(slack_under.value()) + [" "]
style.data.loc["Slack over"] = list(slack_over.value()) + [" "]
display(style)
```

```
Total penalty:  41.2 versus 40.2 before.
```

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan	F	D	D	D	D	D	F	F	D	D	F	F	F
Katherine	D	D	D	D	D	F	F	D	D	F	F	F	D
Robert	D	D	D	D	D	F	F	F	F	D	D	D	F
Jonathan	D	D	F	F	F	D	D	D	D	D	F	F	F
William	F	D	D	D	D	F	F	D	D	F	F	D	D
Richard	D	D	D	F	F	F	D	D	F	F	D	D	F
Kristen	F	F	D	D	D	F	F	D	D	F	F	D	D
Kevin	D	D	F	F	F	F	F	F	F	D	D	D	D
Cover D	5/5	7/7	6/6	5/4	5/5	2/5	2/5	5/6	5/7	4/4	3/2	5/5	4/6
Slack under	0	0	0	0	0	3	3	1	2	0	0	0	2
Slack over	0	0	0	1	0	0	0	0	0	0	1	0	0

Outline of the talk

Part 1: Deductive explanations (What causes X?)

- UNSAT: minimal unsatisfiable subsets
  - efficient MUSes
  - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists

Part 2: Counterfactual explanation (What if Y instead of X?)

- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function ⏪

Counterfactual Explanations for <u>OPT</u> problems

- Corrective actions over the constraints? is UNSAT, get counterfactual explnations from that.

Counterfactual Explanations for OPT problems

- Corrective actions over the constraints? $C \ \wedge \ (f(x) < o)$ is UNSAT, get counterfactual explnations from that.

- Corrective actions over the objective function coefficients:

The user asks: "What coefficients need to change so that Y becomes an optimal solution instead of X?"

Y is a foil from the optimisation perspective: it leads to a non-optimal solution.

> *[Korikov, Anton, and J. Christopher Beck. "Counterfactual explanations via inverse constraint programming." In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021).]*

Counterfactual Explanations for OPT problems

Find currently optimal solution X:

```
In [46]: model, nurse_view = factory.get_full_model()

         assert model.solve()
         print("Total penalty: ", model.objective_value())
         visualize(nurse_view.value(), factory)
```

```
Total penalty:  607
```

Out[46]:

	Week 1							Week 2					
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
name													
Megan	F	D	D	D	D	F	F	D	D	F	F	D	D
Katherine	D	D	D	D	D	F	F	F	D	D	F	F	D
Robert	D	D	D	F	F	D	D	D	F	F	D	D	F
Jonathan	D	D	F	F	F	D	D	D	D	D	F	F	F
William	F	D	D	D	D	F	F	D	D	F	F	D	D
Richard	D	D	D	D	D	F	F	D	D	F	F	D	D
Kristen	F	F	D	D	D	F	F	D	D	D	F	F	D
Kevin	D	D	F	F	F	F	F	F	D	D	D	D	D
Cover D	5/5	7/7	6/6	5/4	5/5	2/5	2/5	6/6	7/7	4/4	2/2	5/5	6/6

Counterfactual Explanations for OPT problems

Robert is unhappy!

```
In [47]: nurse = "Robert"

         for (w,pref) in zip(*model.objective_.args):
             if nurse in str(pref):
                 print(f"{pref.value()} \t w:{w} \t{pref} \t")
```

```
False    w:1    Robert's requests to work shift D on Mon 1 is de
nied
False    w:1    Robert's requests to work shift D on Tue 1 is de
nied
False    w:1    Robert's requests to work shift D on Wed 1 is de
nied
True     w:1    Robert's requests to work shift D on Thu 1 is de
nied
True     w:1    Robert's requests to work shift D on Fri 1 is de
nied
False    w:1    Robert's requests to not work shift D on Sat 2 i
s denied
False    w:1    Robert's requests to not work shift D on Sun 2 i
s denied
```

Counterfactual Explanations for OPT problems

Robert is unhappy!

```
In [47]: nurse = "Robert"

         for (w,pref) in zip(*model.objective_.args):
             if nurse in str(pref):
                 print(f"{pref.value()} \t w:{w} \t{pref} \t")
```

```
False    w:1    Robert's requests to work shift D on Mon 1 is de
nied
False    w:1    Robert's requests to work shift D on Tue 1 is de
nied
False    w:1    Robert's requests to work shift D on Wed 1 is de
nied
True     w:1    Robert's requests to work shift D on Thu 1 is de
nied
True     w:1    Robert's requests to work shift D on Fri 1 is de
nied
False    w:1    Robert's requests to not work shift D on Sat 2 i
s denied
False    w:1    Robert's requests to not work shift D on Sun 2 i
s denied
```

```
In [48]: desc = "Robert's requests to work shift D on Fri 1 is denied"
         weight,d_on_fri1 = next((w,pref) for w,pref in zip(*model.objective_.arg
         print(f"{d_on_fri1.value()} \t w:{w} \t{d_on_fri1}")
```

```
True     w:1    Robert's requests to work shift D on Fri 1 is de
nied
```

Counterfactual Explanations for OPT problems

Robert's request to work on Fri 1 is very important! His daughter has a surgery that day.

How should he minimally change *his* preferences to work that day?

```
In [49]: foil = {d_on_fri1 : False}  # don't want to have his request for Fri 1 d
         print("Foil:", foil, "\n")

         other_prefs = [(w,pref) for w,pref in zip(*model.objective_.args) if nur
         print(f"{nurse}'s other preferences:")
         for w,pref in other_prefs:
             print("- Weight",w,":",pref)
```

```
Foil: {not([roster[2,4] == 1]): False}

Robert's other preferences:
- Weight 1 : Robert's requests to work shift D on Mon 1 is denie
d
- Weight 1 : Robert's requests to work shift D on Tue 1 is denie
d
- Weight 1 : Robert's requests to work shift D on Wed 1 is denie
d
- Weight 1 : Robert's requests to work shift D on Thu 1 is denie
d
- Weight 1 : Robert's requests to not work shift D on Sat 2 is d
enied
- Weight 1 : Robert's requests to not work shift D on Sun 2 is d
enied
```

Counterfactual Explanations for OPT problems

> *[Korikov, Anton, and J. Christopher Beck. "Counterfactual explanations via inverse constraint programming." In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021).]*

Algorithmically, it is a beautiful inverse optimisation problem with a multi-solver main/subproblem algorithm

Counterfactual Explanations for OPT problems

[Korikov, Anton, and J. Christopher Beck. "Counterfactual explanations via inverse constraint programming." In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021).]

Algorithmically, it is a beautiful inverse optimisation problem with a multi-solver main/subproblem algorithm

In [50]:

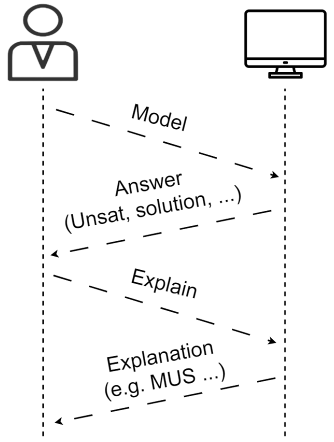
```
from explanations.counterfactual import inverse_optimize

v = model.objective_value()
new_obj = inverse_optimize(model=model, minimize=True,
                           user_sol = foil,
                           allowed_to_change = set(p[1] for p in other_p
print(f"Done! Found solution with total penalty {new_obj.value()}, was {

# Let's look at the preferences he should enter, to avoid Fri 1!
print(f"{nurse} should change the following preferences:")
for w,pref in zip(*new_obj.args):
    if nurse in str(pref) and str(pref) != desc and w != 1:  # previous
        print("- set to weight:", w, "--", pref)
```

```
Done! Found solution with total penalty 607, was 607

Robert should change the following preferences:
- set to weight: 0 -- Robert's requests to not work shift D on S
at 2 is denied
```

Hands-on Explainable Constraint Programming (XCP)

Part 1: Deductive explanations (What causes X?) ✅

- UNSAT: minimal unsatisfiable subsets
  - efficient MUSes
  - preferred MUSes
- SAT: explaining logical consequences
- OPT: explaining that no better solution exists



Part 2: Counterfactual explanation (What if Y instead of X?) ✅

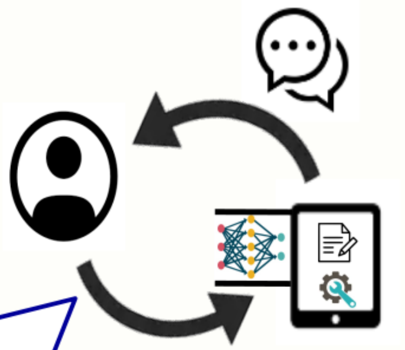
- UNSAT: minimum correction subsets
- UNSAT: corrective actions
- SAT: checking a foil
- OPT: correcting the objective function

Explainable Constraint Programming (XCP)

Recurring challenges:

- **Definition** of explanation: *question and answer format*
- Computational efficiency, **incremental** solvers
- Explanation **selection**: *which explanation to show; learn preferences?*
- User **Interaction**? *(visualisation, conversational, stateful, ...)*
- Explanation **evaluation**: *computational, formal, user survey, user study, ...*

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

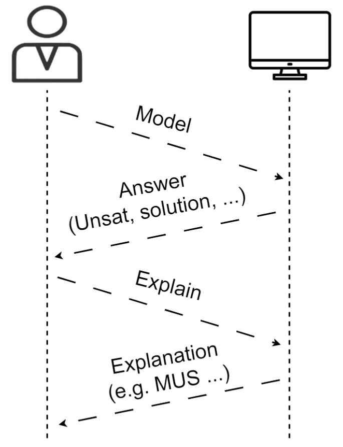
https://people.cs.kuleuven.be/~tias.guns/chat-opt.html

Visitors welcome!

Connections to wider XAI

- Explanations in planning, e.g. MUGS *[Eiflet et al]*, Model Reconciliation *[Chakraborti et al]*, ...
- Explanations for KR/justifications *[Swartout et al]*, ASP *[Fandinno et al]*, in OWL *[Kalyanpur et al]*, ...
- Formal explanations of ML models (e.g. impl. hitting-set based, *[Ignatiev et al]*)

Conclusion (final slide)



- Deductive and Constrastive Explanation of UNSAT/SAT/Opt
- Deductive explanations relate back to finding a MUS/OUS
- XCP requires programmable (multi-solver) tooling (here: CPMpy)

- Many open challenges and new problems!
- Less developed: counterfactual and interactive methods
- We need incremental CP-solvers!

Want to learn more?

Tutorial as notebook available at https://github.com/CPMpy/XCP-explain

(PS. Hiring a post-doc, tell your colleagues to contact me...)

References mentioned (many more exist!!!)

MUS

- Liffiton, M. H., & Sakallah, K. A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning, 40, 1-33.

- Ignatiev, A., Previti, A., Liffiton, M., & Marques-Silva, J. (2015, August). Smallest MUS extraction with minimal hitting set dualization. In International Conference on Principles and Practice of Constraint Programming (pp. 173-182). Cham: Springer International Publishing.

- Joao Marques-Silva. Minimal Unsatisfiability: Models, Algorithms and Applications. ISMVL 2010. pp. 9-14

Feasibility restoration

- Senthooran, I., Klapperstueck, M., Belov, G., Czauderna, T., Leo, K., Wallace, M., ... & De La Banda, M. G. (2021). Human-centred feasibility restoration. In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Explaining optimization problems

- Korikov, A., & Beck, J. C. (2021). Counterfactual explanations via inverse constraint programming. In 27th International Conference on Principles and Practice of Constraint Programming (CP 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Explanation in planning, ASP, KR

- Eifler, Rebecca, Michael Cashmore, Jörg Hoffmann, Daniele Magazzeni, and Marcel Steinmetz. "A new approach to plan-space explanation: Analyzing plan-property dependencies in oversubscription planning." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, no. 06, pp. 9818-9826. 2020.
- Chakraborti, Tathagata, Sarath Sreedharan, Yu Zhang, and Subbarao Kambhampati. "Plan explanations as model reconciliation: moving beyond explanation as soliloquy." In Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 156-163. 2017.
- Fandinno, Jorge, and Claudia Schulz. "Answering the “why” in answer set programming–A survey of explanation approaches." Theory and Practice of Logic Programming 19, no. 2 (2019): 114-203.
- Swartout, William, Cecile Paris, and Johanna Moore. "Explanations in knowledge systems: Design for explainable expert systems." IEEE Expert 6, no. 3 (1991): 58-64.
- Kalyanpur, Aditya, Bijan Parsia, Evren Sirin, and Bernardo Cuenca-Grau. "Repairing unsatisfiable concepts in OWL ontologies." In The Semantic Web: Research and Applications: 3rd European Semantic Web Conference, ESWC 2006 Budva, Montenegro, June 11-14, 2006 Proceedings 3, pp. 170-184. Springer Berlin Heidelberg, 2006.

Formal explantions in ML

- Ignatiev, Alexey, Nina Narodytska, and Joao Marques-Silva. "Abduction-based explanations for machine learning models." In Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, pp. 1511-1519. 2019.