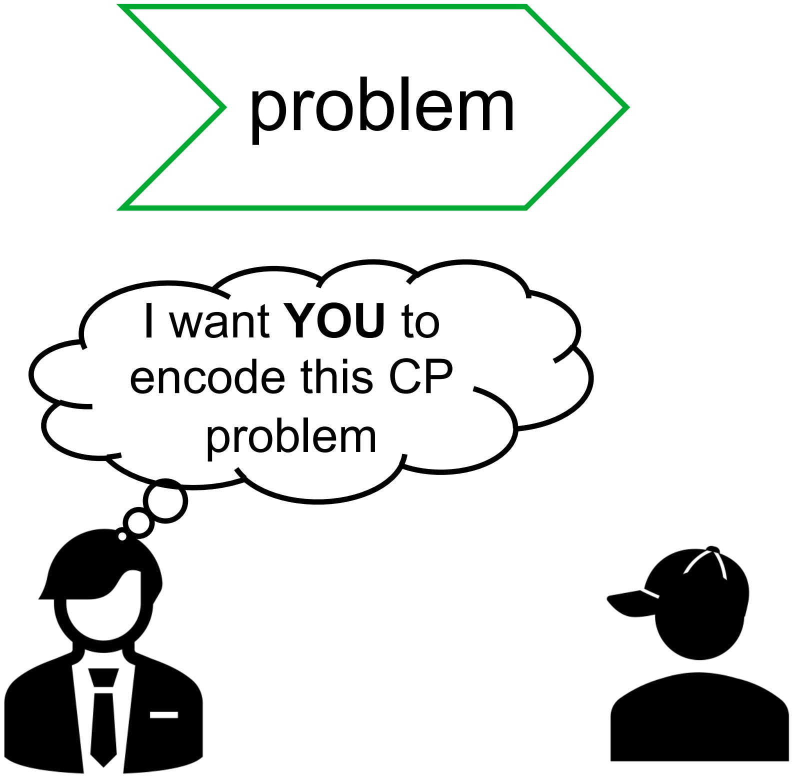
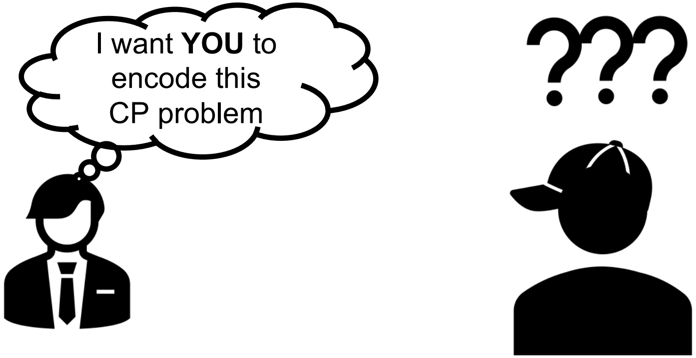
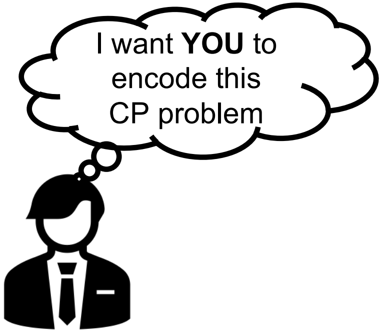
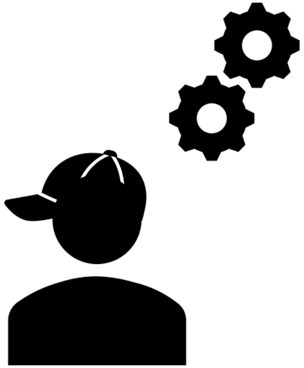

problem
I want **YOU** to encode this CP problem

problem

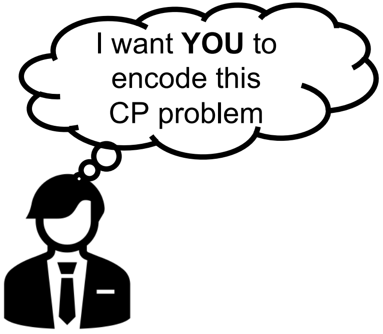


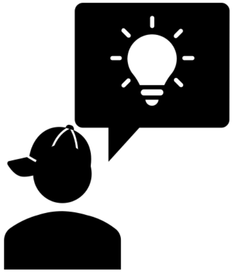
problem model



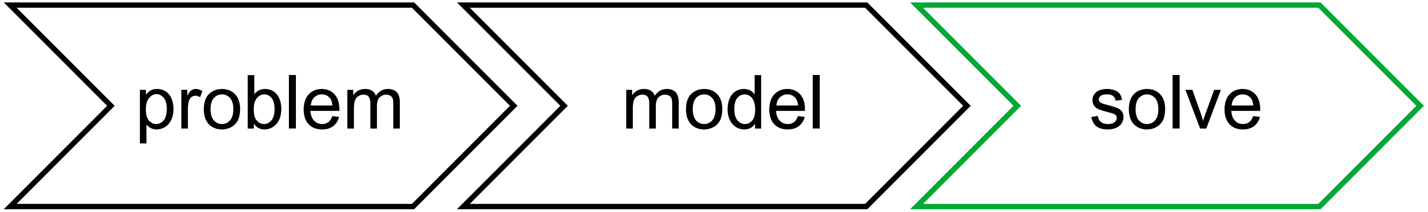


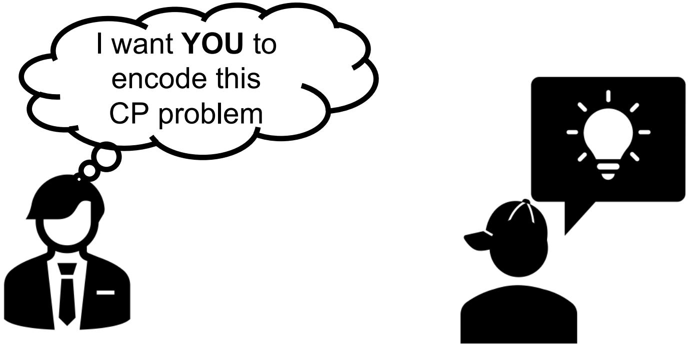
problem → model



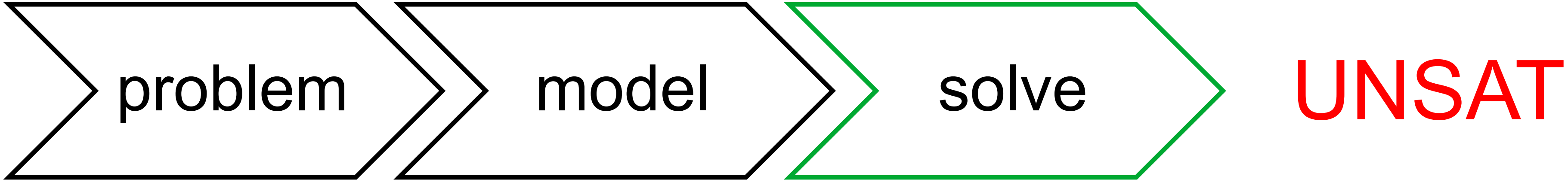


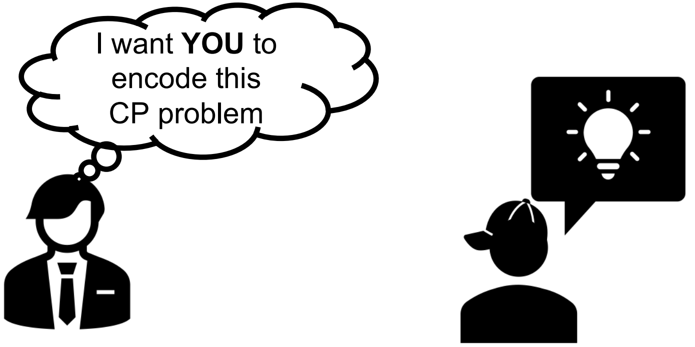
```
x = intvar(-9, 9, name="x")
y = intvar(-9, 9, name="y")
m = Model(
    x < 0,
    x < 1,
    x > 2,
    (x + y > 0) | (y < 0),
    (y >= 0) | (x >= 0),
    (y < 0) | (x < 0),
    (y > 0) | (x < 0),
    AllDifferent(x,y)
)
```






```
x = intvar(-9, 9, name="x")
y = intvar(-9, 9, name="y")
m = Model(
    x < 0,
    x < 1,
    x > 2,
    (x + y > 0) | (y < 0),
    (y >= 0) | (x >= 0),
    (y < 0) | (x < 0),
    (y > 0) | (x < 0),
    AllDifferent(x,y)
)
```






```
x = intvar(-9, 9, name="x")
y = intvar(-9, 9, name="y")
m = Model(
    x < 0,
    x < 1,
    x > 2,
    (x + y > 0) | (y < 0),
    (y >= 0) | (x >= 0),
    (y < 0) | (x < 0),
    (y > 0) | (x < 0),
    AllDifferent(x,y)
)
```

Solver says UNSAT, what now?

- (Human) modeling error ?
- Problem is <u>over-constrained </u>or <u>unsatisfiable</u> ?

Talk consists of

1. Using Constraint Solvers as an oracle, with CPMpy

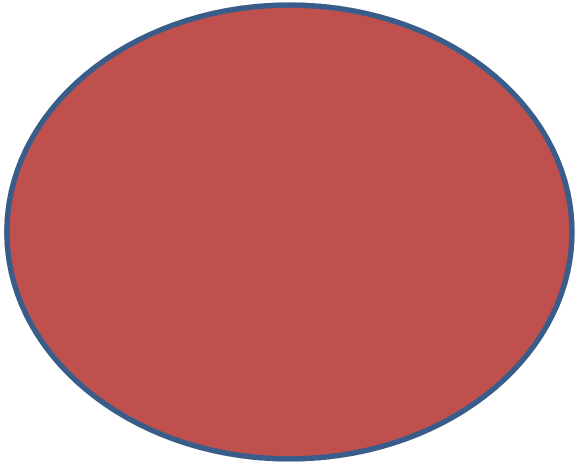
2. **Explaining (un)satisfiabilty: examples of master/sub-problem solving**

3. Advanced examples: Explaining Optimality (using logic cutting-planes)

Conclusion, outlook and questions

Debugging a model

= Need for an explanation of UNSAT



Model

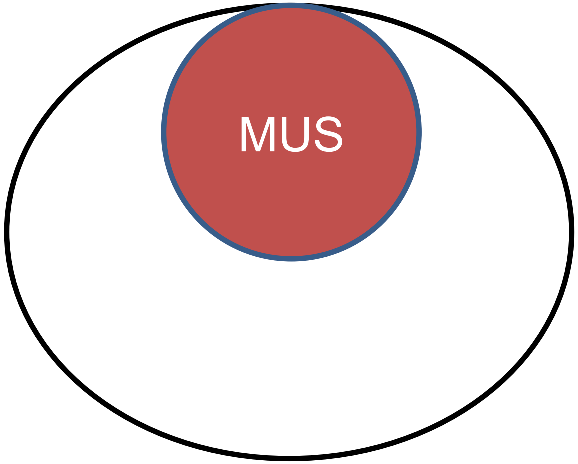
Debugging a model

= Need for an explanation of UNSAT

1. Identify conflicting constraints as explanation for UNSAT

→ Extract Minimum Unsatisfiable Subset (MUS)

a.k.a Irreducible Inconsistent Subsystem (IIS)



Model

Debugging a model

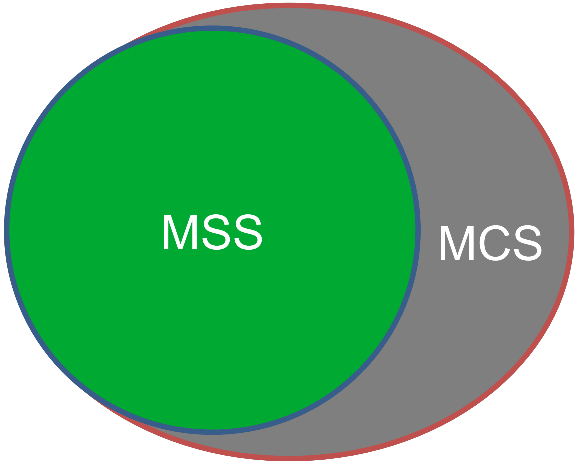
= Need for an explanation of UNSAT

1. Identify conflicting constraints as explanation for UNSAT

   → Extract Minimum Unsatisfiable Subset (MUS)

   a.k.a Irreducible Inconsistent Subsystem (IIS)

2. Identify Maximal Satisfiable Subset (MSS)



Debugging a model

= Need for an explanation of UNSAT

1. Identify conflicting constraints as explanation for UNSAT

   → Extract a Minimum Unsatisfiable Subset (MUS)

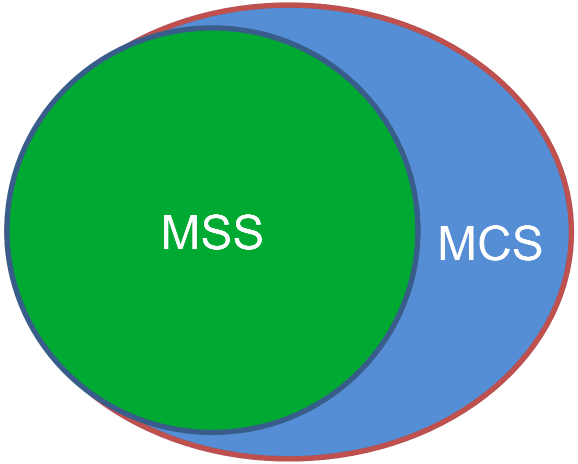
   a.k.a Irreducible Inconsistent Subsystem (IIS)

2. Identify Maximal Satisfiable Subset (MSS)

3. “Correct” the infeasibility in the model

   → Extract Minimum Correction Subsets (MCS)

   Complement of some MSS, removal/correction leads to a satisfiable subset



Debugging a model

= Need for an explanation of UNSAT

1. Identify <u>conflicting constraints</u> as explanation for UNSAT

   → Extract <u>Minimum</u> Unsatisfiable Subset (MUS)

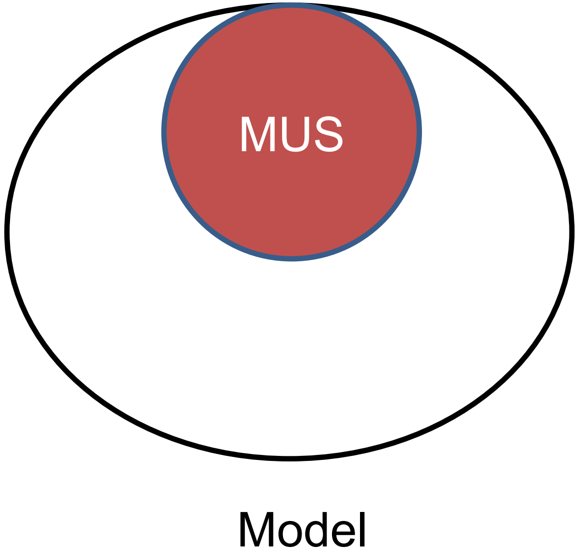
   a.k.a Irreducible Inconsistent Subsystem (IIS)

2. Identify Maximal Satisfiable Subset (MSS)

3. "Correct" the infeasibility in the model

   → Extract Minimum Correction Subsets (MCS)

   Complement of some MSS, removal/correction leads to a satisfiable subset



Explaining UNSAT with MUSes

Methods

1. Some solvers provide an implementation for extracting unsatisfiable cores as explanations of UNSAT.

2. **Deletion-based** Minimal unsatisfiable subsets
   - Iterate over constraints
   - Delete constraints if removing them leaves the model UNSAT

```
def mus(constraints):
    m = Model(constraints)
    assert ~m.solve(), "MUS: model must be UNSAT"

    core = m.get_core()  # or all constraints                 1
    i = 0
    while i < len(core):
        subcore = core[:i] + core[i+1:]  # check if all but i makes core SAT

        if Model(subcore).solve():
            i += 1  # removing it makes it SAT, must keep
        else:                                                     2
            core = subcore  # overwrite core, so core[i] is next one

    return core
```

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

Example of MUS extraction

examples/tutorial_ijcai22/3_musx.ipynb

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```
SAT so in MUS: (x[0]) > (x[1])
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```
SAT so in MUS: (x[0]) > (x[1])
SAT so in MUS: (x[1]) > (x[2])
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```
SAT so in MUS: (x[0]) > (x[1])
SAT so in MUS: (x[1]) > (x[2])
SAT so in MUS: (x[2]) > (x[0])
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```
SAT so in MUS: (x[0]) > (x[1])
SAT so in MUS: (x[1]) > (x[2])
SAT so in MUS: (x[2]) > (x[0])
UNSAT so not in MUS: (x[3]) > (x[0])
```

```python
x = intvar(0,3, shape=4, name="x")
# circular 'bigger than', UNSAT
mus_cons = [
    x[0] > x[1],
    x[1] > x[2],
    x[2] > x[0],

    x[3] > x[0],
    (x[3] > x[1]).implies(x[3] > x[2]) & ((x[3] == 3) | (x[1] == x[2]))
]
```

```python
core = m.get_core()  # or all constraints
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

```
SAT so in MUS: (x[0]) > (x[1])
SAT so in MUS: (x[1]) > (x[2])
SAT so in MUS: (x[2]) > (x[0])
UNSAT so not in MUS: (x[3]) > (x[0])
UNSAT so not in MUS: (((x[3]) > (x[1])) -> ((x[3]) > (x[2]))) and ((x[3] == 3) or ((x[1]) == (x[2])))
```

Explaining UNSAT with MUSes

Methods & Insights

- Some solvers provide unsatisfiable cores as a starting point for debugging
- Deletion-based Minimal unsatisfiable subsets

KEY Insights

✓ Faster if the solver supports unsat core extraction and assumptions especially for larger problems

- ❑ Most solvers provide an assumption interface
- ❑ Does not require many modifications

X Depends on the ordering of the clauses

```python
i = 0 # we will dynamically shrink mus vars
while i < len(core):
    # add all other remaining constraints
    subcore = core[:i] + core[i+1:]

    if Model(subcore).solve():
        # with all but 'i' it is SAT, so 'i' belongs to the MUS
        print("\tSAT so in MUS", core[i])
        i += 1
    else:
        # still UNSAT, 'i' does not belong to the MUS
        print("\tUNSAT so not in MUS", core[i])
        # overwrite current 'i' and continue
        core = subcore
```

Example of using assumptions for MUS extraction

examples/tutorial_ijcai22/3_musx.ipynb

Explaining UNSAT with MUSes

Methods & Insights

- Some solvers provide unsatisfiable cores as a starting point for debugging.
- Deletion-based Minimal unsatisfiable subsets

KEY Insights

✓ Depends on the ordering of the clauses

✓ Faster if the solver supports unsat core extraction and assumptions especially for larger problems

- ❑ Most solvers provide an assumption interface
- ❑ Does not require many modifications

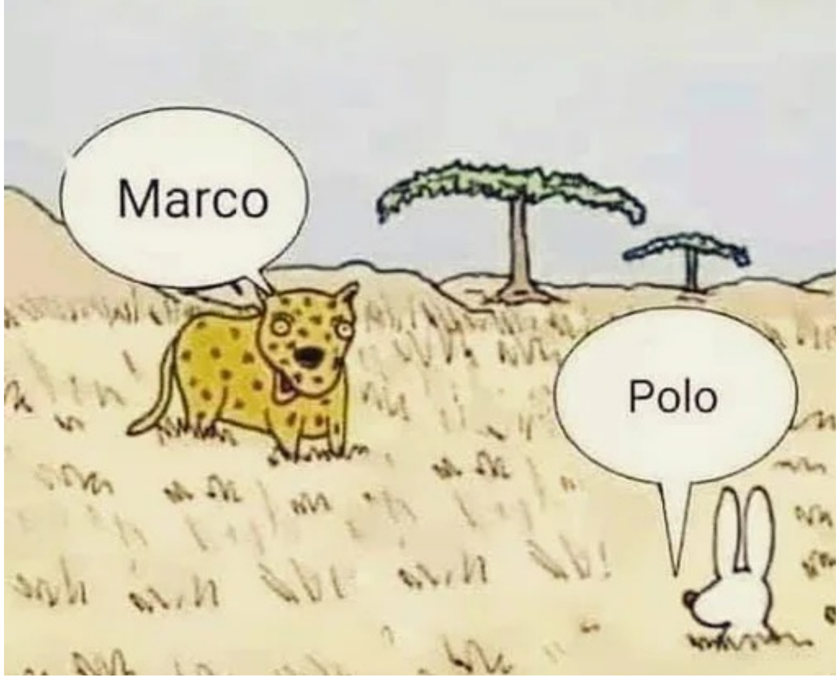
Enumerate all Minimal Unsatisfiable Subsets and Maximal Satisfiable Subsets

Master ←→ sub-problem

Marco Polo
(1254 –1324)

Marco-Polo

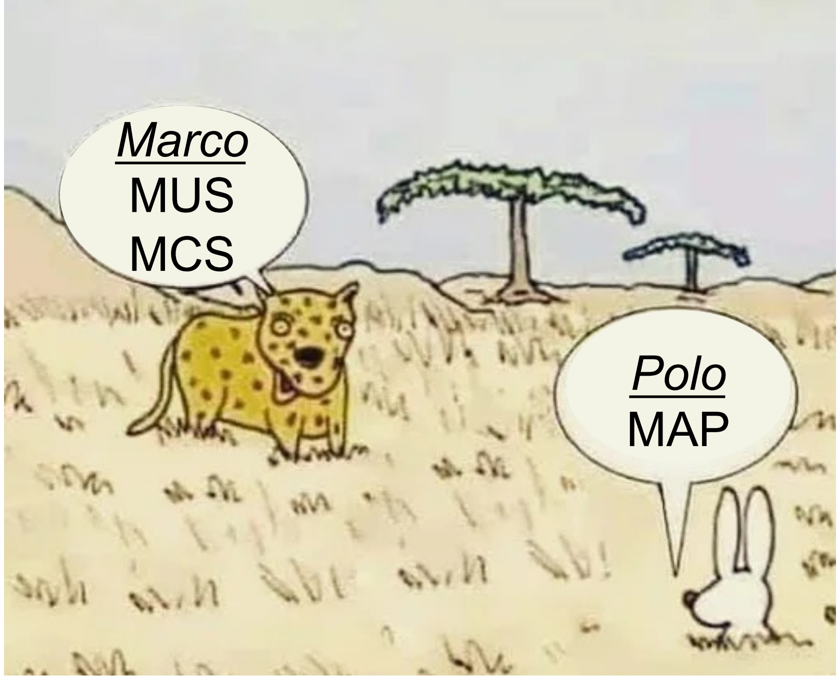
An example of Master ←→ Sub-problem approach



Source: https://9gag.com/gag/a07MMXZ

Marco-Polo

An example of Master ←→ Sub-problem approach



MARCO: **Ma**pping **R**egions of **Co**nstraints sets

- MUS enumeration algorithm

POLO: **Po**wer **Lo**gic

- "Map" of the powerset as a propositional logic Formula

[Liffiton, Mark H., and Ammar Malik. (2013) "Enumerating infeasibility: Finding multiple MUSes quickly." *CPAIOR 2013*.]

Source: https://9gag.com/gag/a07MMXZ

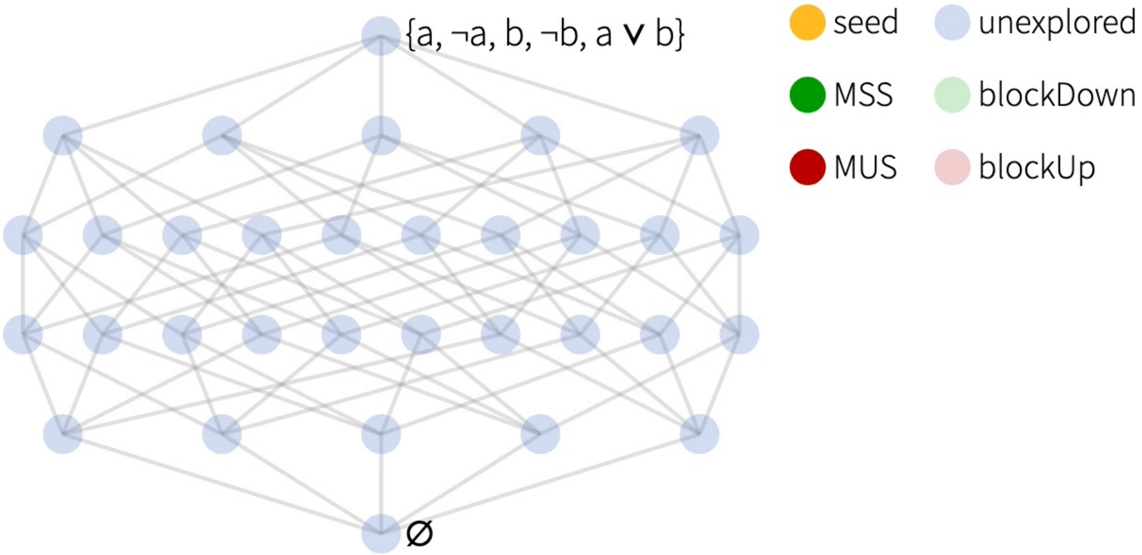
MARCO Algorithm

Input: Constraint system C

```
Map ← T
while Map is satisfiable:
    seed ← getUnexplored(Map)
    if seed is satisfiable:
        MSS ← grow(seed, C)
        output MSS
        Map ← Map ∧ blockDown(MSS)
    else:
        MUS ← shrink(seed, C)
        output MUS
        Map ← Map ∧ blockUp(MUS)
```

Example input: $C = \{a, \neg a, b, \neg b, a \vee b\}$

seed :
Map :
MUSes :
MSSes :



Source: https://sun.iwu.edu/~mliffito/marco-viz/

MarcoPolo MUS/MSS enumeration

examples/tutorial_ijcai22/4_marco-mus-mcs-enumeration.ipynb

Explaining UNSAT with MUSes

Methods & Insights

- Deletion-based MUS extracts only 1MUS
- Efficiently enumerate all Minimal Unsatisfiable Subsets and Maximal Satisfiable Subsets
- No guarantee of cardinal-minimality (only subset minimal)

  → Smallest Minimal Unsatisfiable Subset (SMUS)

  or optimality with weighted constraints

  → Optimal (Constrained) Unsatisfiable Subset (OCUS)

Master ←→ Sub-problem

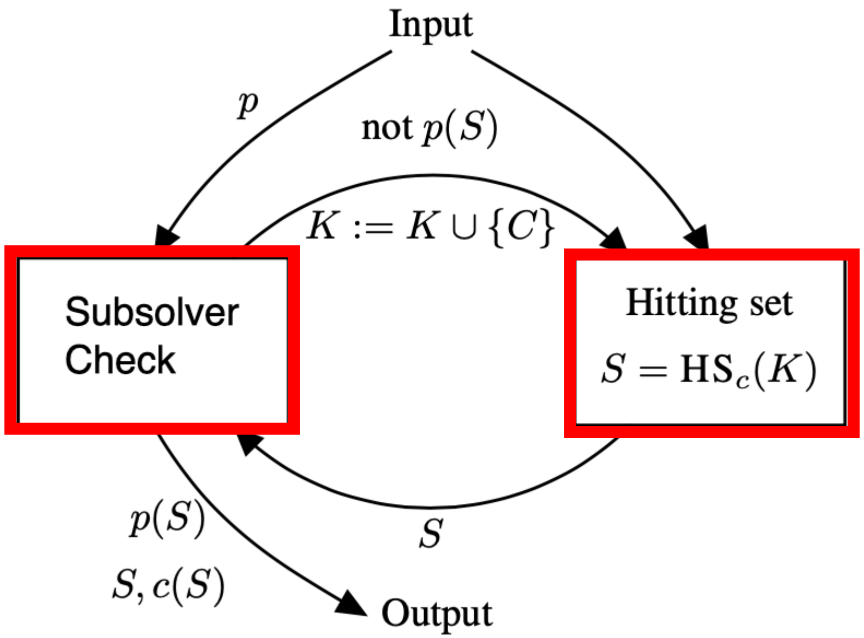
Implicit hitting set algorithms

Master ←→ Sub-problem approach

Implicit hitting set algorithms

General Structure:

1. Find minimum hitting set
2. Call subsolver for checking (SAT, UNSAT ….)



Saikko, Paul, Johannes P. Wallner, and Matti Järvisalo. "Implicit hitting set algorithms for reasoning beyond NP." *Fifteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2016.

Master ←→ Sub-problem approach

Implicit hitting set algorithms

***Smallest MUS* [1] and *Optimal (C)US* [2]:**

- Deciding whether a MUS of size ≤ k is $\sum_p^2 -complete$
- Extracting a smallest MUS (OCUS/SMUS) is in $FP^{\Sigma_p^2}$

Based on the implicit hitting set duality between **MCSs** and **MUSs**:

- Also used for MaxSAT, the dual of the OCUS-problem, i.e. MaxHS [3]

A set S ⊆ F is a MCS of F if and only if it is a *minimum hitting set* of MUSs(F).

A set S ⊆ F is a MUS of F if and only if it is a *minimum hitting set* of MCSs(F).

[1] E. Gamba, B. Bogaerts, T. Guns, Efficiently Explaining CSPs with Unsatisfiable Subset Optimization, IJCAI 2021
[2] A. Ignatiev, et al. "Smallest MUS extraction with minimal hitting set dualization." CP 2015.
[3] J. Davies, and B.Fahiem. "Exploiting the power of MIP solvers in MAXSAT." SAT 2013.

Optimal Constrained Unsatisfiable Subsets

Implicit hitting set-based algorithm

→ uses an implicit hitting set algorithm (like SMUS and MaxHS)

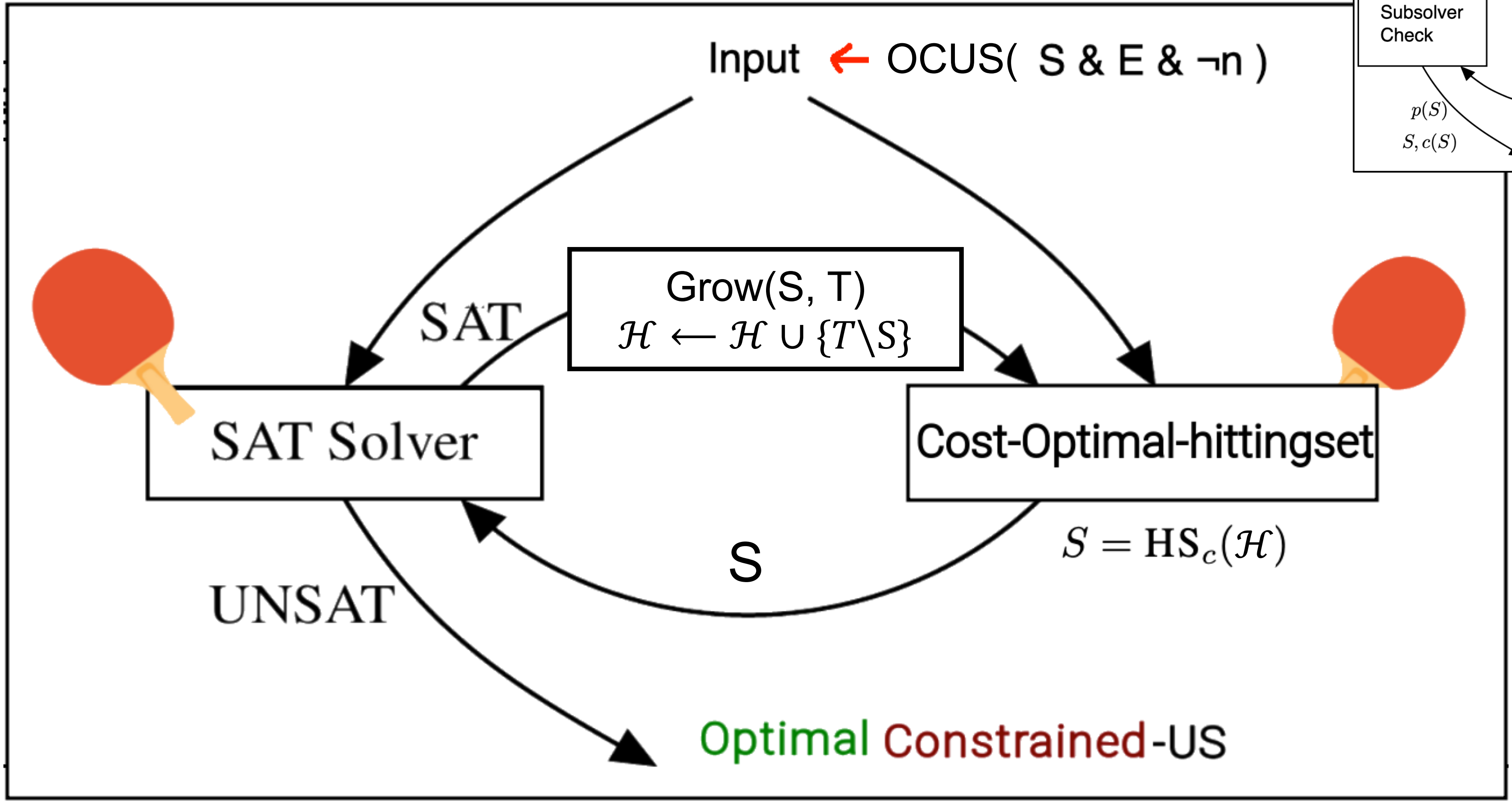
Algorithm 4: OCUS(T, f, p)

$\mathcal{H} \leftarrow \emptyset$
while true **do**
  $\mathcal{S} \leftarrow$ COST-OPTIMAL-HITTINGSET$(\mathcal{H}, f, p)$
  if $\neg$SAT$(\mathcal{S})$ **then**
    return $\mathcal{S}$
  end
  $\mathcal{S} \leftarrow$ GROW$(\mathcal{S}, T)$
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{T \setminus \mathcal{S}\}$
end

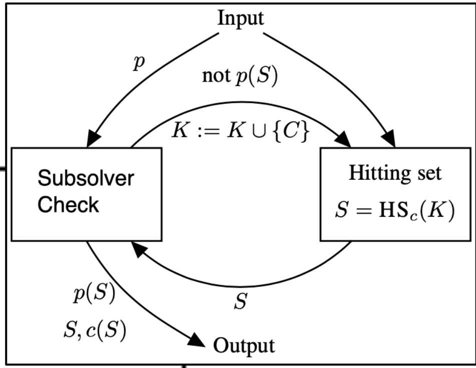
MIP solver

SAT/CP solver
as an oracle

[E. Gamba, B. Bogaerts, T. Guns, Efficiently Explaining CSPs with Unsatisfiable Subset Optimization, IJCAI 2021]



Input ← OCUS(S & E & ¬n)
SAT
Grow(S, T)
$\mathcal{H} \longleftarrow \mathcal{H} \cup \{T \backslash S\}$
SAT Solver
Cost-Optimal-hittingset
$S = \mathrm{HS}_c(\mathcal{H})$
S
UNSAT
Optimal Constrained-US



Input
p
not $p(S)$
$K := K \cup \{C\}$
Subsolver Check
Hitting set
$S = \mathrm{HS}_c(K)$
$p(S)$
$S, c(S)$
S
Output

OCUS *with* assumptions

```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)
```

Hitting set Solver

repeatedly compute hitting sets

CP/SAT as an oracle

Extract Correction Subset

OUS extraction

examples/tutorial_ijcai22/5_ocus_explanations.ipynb

Optimal Constrained Unsatisfiable Subsets

Implicit hitting set-based algorithm

- Example of multi-solver incremental solving
- *Made easier and efficient with assumptions*

1. **Need to repeatedly compute hitting sets.**
   - Problem becomes hard as collection of sets-to-hits <u>expands</u>.
   - Big efficiency gains if incremental (and not restart)!

2. **CP/SAT is used as an oracle**
   - (OCUS) CP/SAT checking satisfiability of a subset
   - (OCUS) Grow solves a MaxSAT problem, s.t. complement is a (small) MCS

[E. Gamba, B. Bogaerts, T. Guns, Efficiently Explaining CSPs with Unsatisfiable Subset Optimization, IJCAI 2021]

Talk consists of

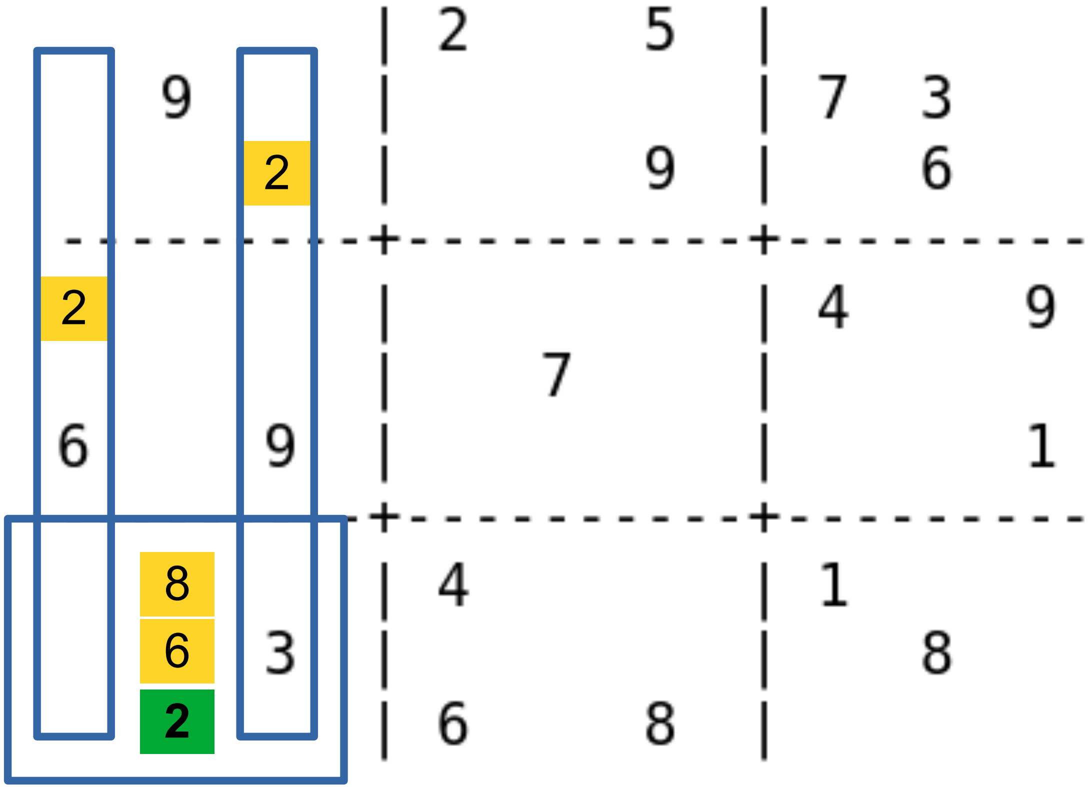
1. Using Constraint Solvers as an oracle, with CPMpy
2. **Explaining (un)satisfiabilty: examples of master/sub-problem solving**
3. Advanced examples: Explaining Optimality (using logic cutting-planes)

Conclusion, outlook and questions

What if a model is SAT?

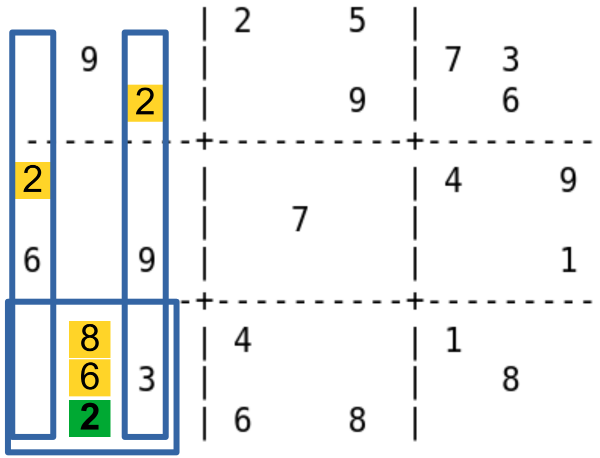
```
      | 2   5 |
  9   |       | 7 3
    2 |     9 |   6
------+-------+------
2     |       | 4   9
      |   7   |
6   9 |       |     1
------+-------+------
  8   | 4     | 1
  6 3 |       |   8
      | 6   8 |
```

What if a model is SAT?



What if a model is SAT?

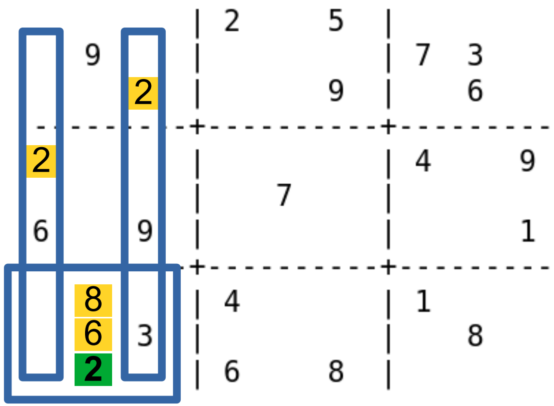
- User may not understand all derivations
- Or wants to learn about it



***“Explain** in a human-understandable way how to solve constraint satisfaction problems”*

Explanation step

Let *E’* & *S’* => ***n*** be one explanation step.



E’ = a subset of previously derived facts E

(Sudoku) Given and derived digits in the grid

S’ = a minimal subset of constraints S such that E’ & S’ => n

(Sudoku) Alldifferent column, row, box constraints

n = a newly derived fact (from the solution)

How? MUS(¬ n & E & S) is a valid explanation step

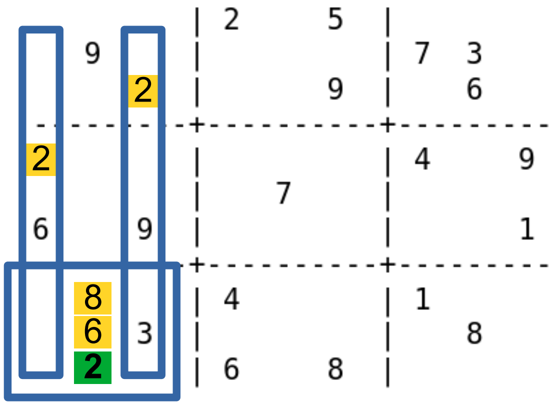
The best/easiest explanation step...

Let *f(S)* be a *cost function* that quantifies how good (e.g. easy to understand) an explanation step is.

Simple MUS-based algo:

```
sol-to-explain = propagate( E & S ) \ E

X_best = None
for n in sol-to-explain:
    X = MUS(~n & E & S )
    if f(X) < f(X_best):
        X_best = X
return X_best
```



MUS gives no guarantees on quality, only subset minimal (SMUS)

The best/easiest explanation step...

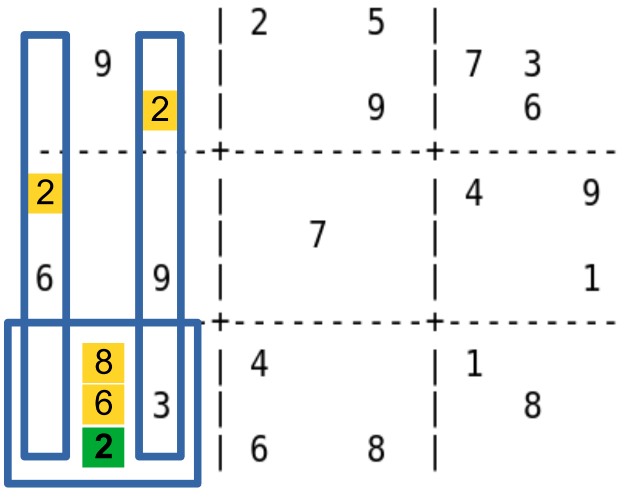
Let *f(S)* be a *cost function* that quantifies how good (e.g. easy to understand) an explanation step is.

Explain 1 step with OCUS

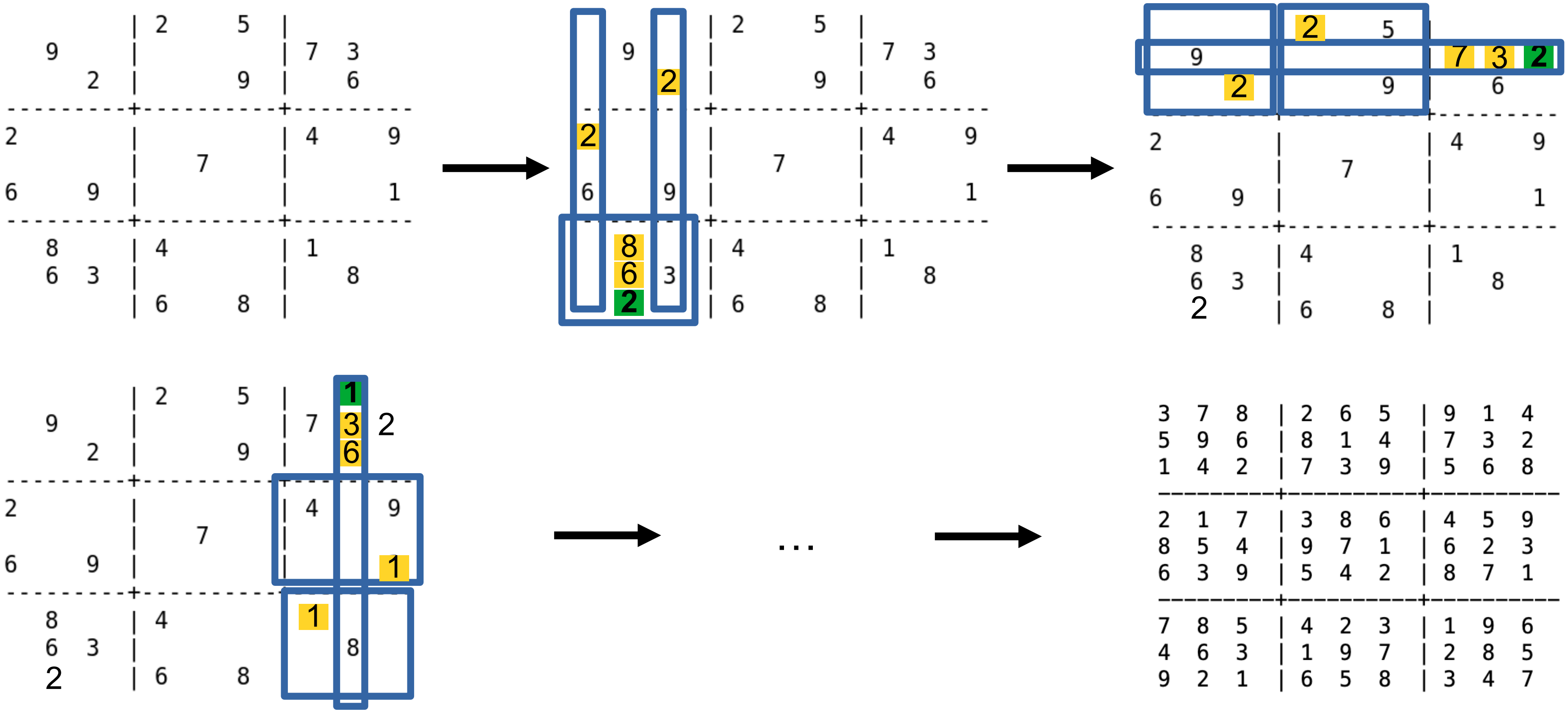
sol-to-explain = propagate(E & S) \ E

p = exactly-one({~n|n ∈ *sol-to-explain*}),

return OCUS(n|n ∈ sol-to-explain} & S & E &{~,**f**, **p**)



A sequence of explanations to explain SAT



OCUS for explaining SUDOKU

Demo

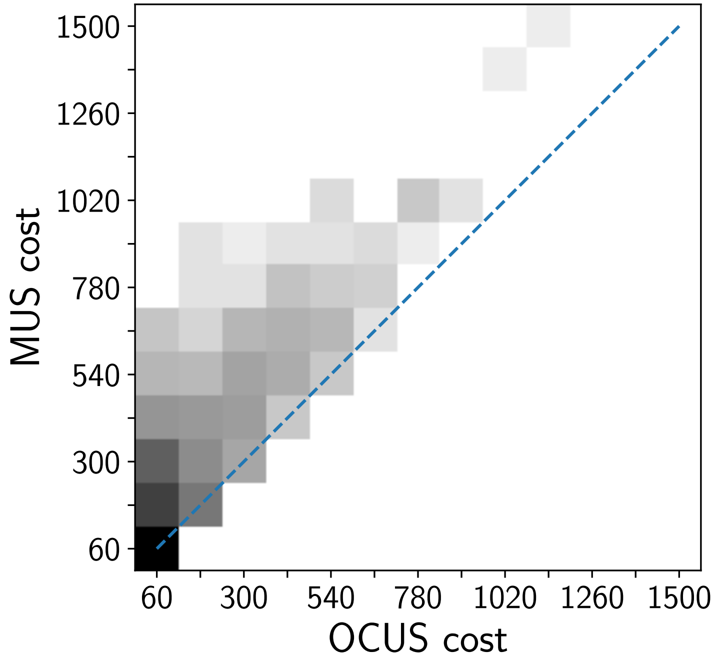
examples/tutorial_ijcai22/6_explain_sudoku.ipynb

The best/easiest explanation step...

```
sol-to-explain = propagate( E & S ) \ E
```

```
X_best = None
for n in sol-to-explain:
    X = MUS(~n & E & S )
    if f(X) < f(X_best):
        X_best = X
return X_best
```

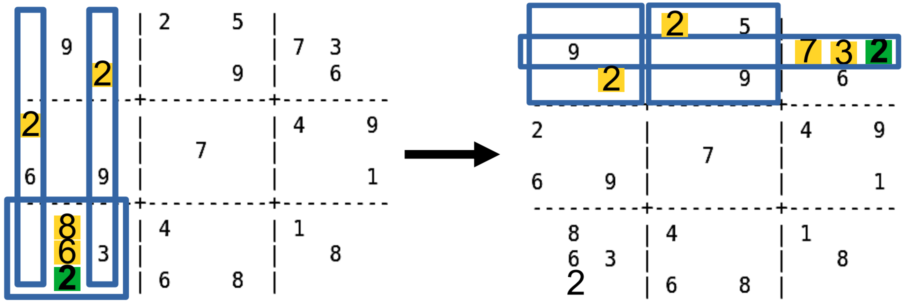
p = exactly-one({~n | n ∈ sol-to-explain}),
OCUS({~n | n ∈ sol-to-explain} & E & S, f, p)



Gamba, E., Bogaerts, B., & Guns, T. (8 2021). Efficiently Explaining CSPs with Unsatisfiable Subset Optimization. In Z.-H. Zhou (Red), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21 (bll 1381–1388). doi:10.24963/ijcai.2021/191.

Incrementality at the Sequence-level

$$\mathbf{OCUS}(\ \boxed{S}\ \&\ E\ \&\neg\ n\ , f, p)$$



S Model constraints

- ➢ Do not change from an explanation step to another

E Derived facts of E

- ➢ Precision-increasing!

Incrementality at the Sequence-level

In practice

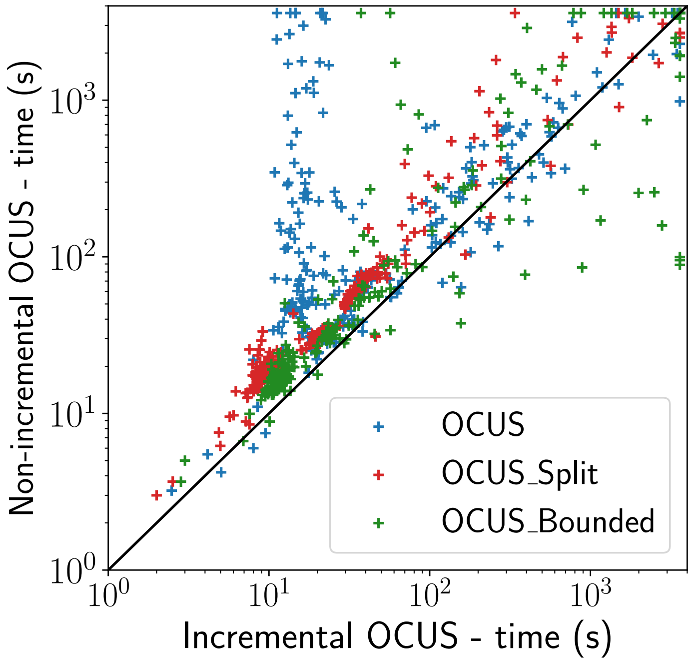
OCUS(S & *E* &¬ ***n*** , *f* , *p* **)**

Incremental OCUS works with the full unsatisfiable formula of step 0

S & E_{end} & {¬ ***n*** | ***n*** ∈ sol-to-explain}

Initialize hitting set solver **once** and modify objective at every explanation step *i* such that

- Underived facts cannot be taken
- Negated facts (¬ **n**) already explained should <u>not</u> be selected
- Assumptions are used to deactivate unused clauses



Summary observations

(OCUS) Multi-solver Incremental solving

Multi-solver

- MIP – highly effective solving hittingset problem
- CP/SAT is used as an oracle for checking satisfiability of a subset

Incremental

- Need to *repeatedly compute hitting sets*.
- Problem becomes *hard* as collection of sets-to-hits expands.
- Big efficiency gains if incremental (and not restart)!
- *(Sequence)* Sets-to-hit re-used between explanation steps