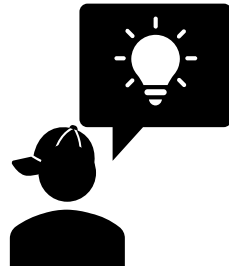
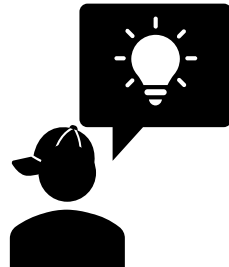
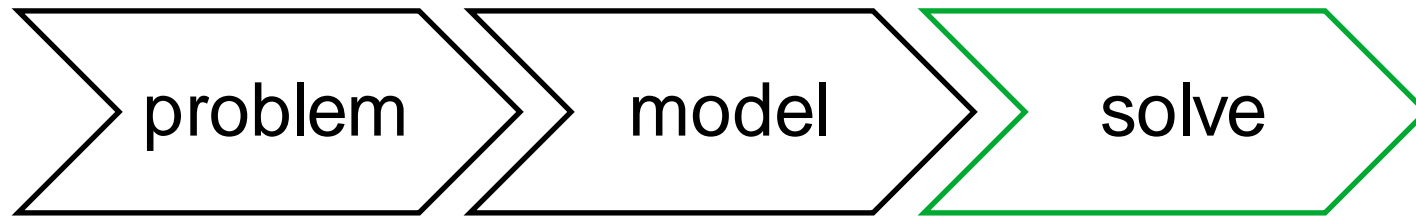


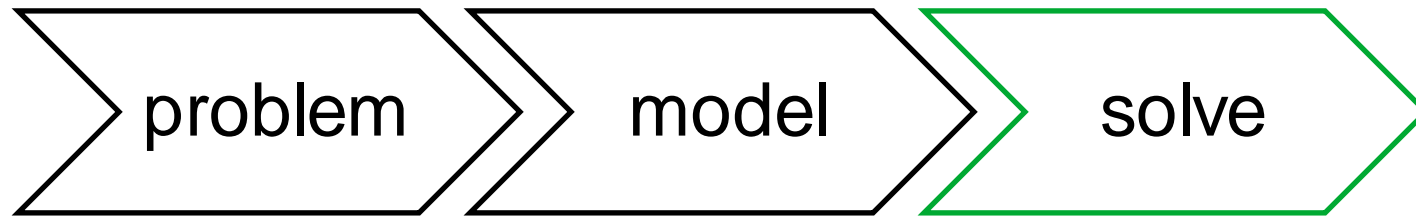


```
x = intvar(-9, 9, name="x")
y = intvar(-9, 9, name="y")
m = Model(
    x < 0,
    x < 1,
    x > 2,
    (x + y > 0) | (y < 0),
    (y >= 0) | (x >= 0),
    (y < 0) | (x < 0),
    (y > 0) | (x < 0),
    AllDifferent(x,y)
)
```

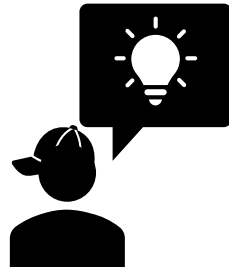




```
x = intvar(-9, 9, name="x")
y = intvar(-9, 9, name="y")
m = Model(
    x < 0,
    x < 1,
    x > 2,
    (x + y > 0) | (y < 0),
    (y >= 0) | (x >= 0),
    (y < 0) | (x < 0),
    (y > 0) | (x < 0),
    AllDifferent(x,y)
)
```



UNSAT



```
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?

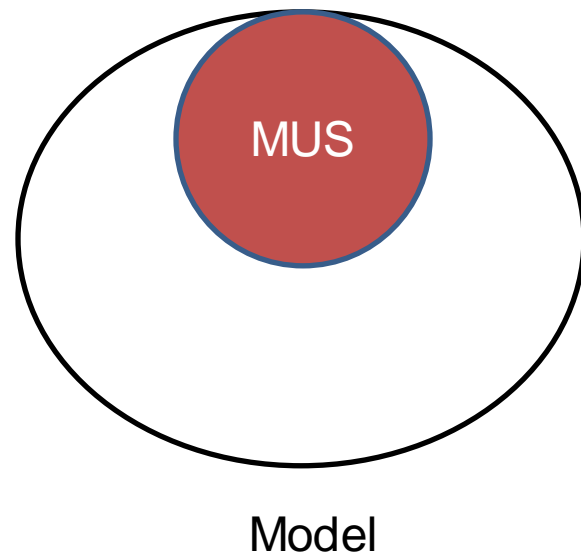
- (Human) modeling error ?
- Problem is over-constrained or unsatisfiable ?

Part 2: Explaining Unsatisfiability

with examples of Master \leftrightarrow Sub-problem solving

Debugging a model

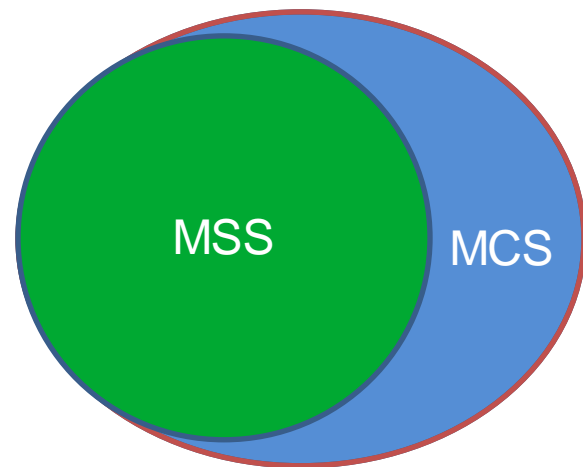
- Need for an explanation of UNSAT
 1. Identify conflicting constraints as explanation for UNSAT
 - Extract Minimum Unsatisfiable Subset (MUS)
a.k.a UNSAT Core, Irreducible Inconsistent Subsystem (IIS)
 2. Identify Maximal Satisfiable Subset (MSS)
 3. “Correct” the infeasibility in the model
 - Extract Minimum Correction Subsets (MCS)
Complement of some MSS, removal/correction leads to a satisfiable subset



Debugging a model

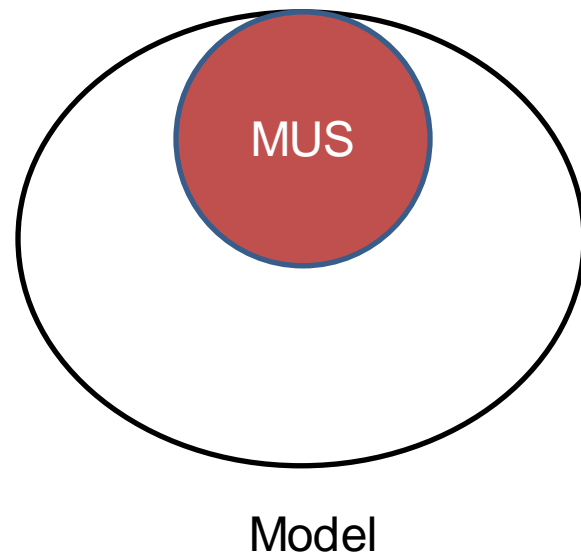
= Need for an explanation of UNSAT

1. Identify conflicting constraints as explanation for UNSAT
→ Extract Minimum Unsatisfiable Subset (MUS)
a.k.a UNSAT Core, 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 conflicting constraints as explanation for UNSAT
 - Extract Minimum Unsatisfiable Subset (MUS)
a.k.a UNSAT Core, 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

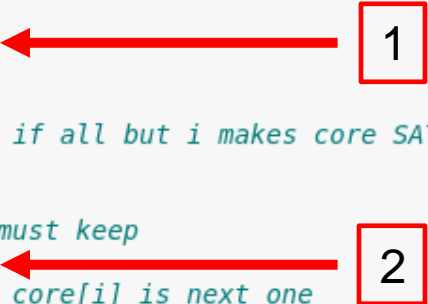


Explaining UNSAT with MUSes

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  
    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:  
            core = subcore # overwrite core, so core[i] is next one  
  
    return core
```



1

2

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

Example of MUS extraction

[cpmpy/examples/tutorial_ijcai22/2_musx_ocus.ipynb](#)

```
x = intvar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]
```

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

```

x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]

```

```

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

```

```
x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]
```

```
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: $(x[0]) > (x[1])$



```
x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]
```

```
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])
```



```
x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]
```

```
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])
```

```

x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]

```

```

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])

```

```

x = intvar(0,3, shape=4, name="x")
# circular 'bigger then', 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]))
]

```

```

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: $((x[3]) > (x[1])) \rightarrow ((x[3]) > (x[2]))$ and $((x[3] == 3) \text{ or } ((x[1]) == (x[2])))$

Explaining UNSAT with MUSes

Methods & Insights

- Some solvers provide unsatisfiable cores as a starting point for debugging.
- Deletion-based Minimal unsatisfiable subsets
 - Iterate over constraints
 - Delete constraints if removing them leaves the model UNSAT

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

```

x = intvar(0,3, shape=4, name="x")
# circular 'bigger then', 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
]

```

```

assum_model = Model()
# make assumption indicators, add reified constraints
ind = BoolVar(shape=len(mus_cons), name="ind")
for i,bv in enumerate(ind):
    assum_model += [bv.implies(mus_cons[i])]
# to map indicator variable back to soft_constraints
indmap = dict((v,i) for (i,v) in enumerate(ind))

assum_solver = CPM_ortools(assum_model)
assert (not assum_solver.solve(assumptions=ind)), "Model must be UNSAT"

```

```

x = intvar(0,3, shape=4, name="x")
# circular 'bigger then', 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
]

```

```

assum_model = Model()
# make assumption indicators, add reified constraints
ind = BoolVar(shape=len(mus_cons), name="ind")
for i,bv in enumerate(ind):
    assum_model += [bv.implies(mus_cons[i])]
# to map indicator variable back to soft_constraints
indmap = dict((v,i) for (i,v) in enumerate(ind))

assum_solver = CPM_ortools(assum_model)
assert (not assum_solver.solve(assumptions=ind)), "Model must be UNSAT"

# unsat core is an unsatisfiable subset
mus_vars = assum_solver.get_core()
print("UNSAT core of size", len(mus_vars))

```



```

x = IntVar(0,3, shape=4, name="x")
# circular 'bigger then', 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
]

```

```

assum_model = Model()
# make assumption indicators, add reified constraints
ind = BoolVar(shape=len(mus_cons), name="ind")
for i,bv in enumerate(ind):
    assum_model += [bv.implies(mus_cons[i])]
# to map indicator variable back to soft_constraints
indmap = dict((v,i) for (i,v) in enumerate(ind))

assum_solver = CPM_ortools(assum_model)
assert (not assum_solver.solve(assumptions=ind)), "Model must be UNSAT"

# unsat core is an unsatisfiable subset
mus_vars = assum_solver.get_core()
print("UNSAT core of size", len(mus_vars))

```

```

# now we shrink the unsatisfiable subset further
i = 0 # we will dynamically shrink mus_vars
while i < len(mus_vars):
    # add all other remaining constraints
    assum_vars = mus_vars[:i] + mus_vars[i+1:]

```

```

if assum_solver.solve(assumptions=assum_vars):
    # with all but 'i' it is SAT, so 'i' belongs to the MUS
    print("\tSAT so in MUS:", mus_cons[i])
    i += 1

```

```

else:
    # still UNSAT, 'i' does not belong to the MUS
    print("\tUNSAT so not in MUS:", mus_cons[i])
    # overwrite current 'i' and continue
    mus_cons = testcons

```

UNSAT core of size 3

```

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

```

Also use new core here

Explaining UNSAT with MUSes

Methods & Insights

- Some solvers provide unsatisfiable cores as a starting point for debugging.
- Deletion-based Minimal unsatisfiable subsets
 - Iterate over constraints
 - Delete constraints if removing them leaves the model UNSAT

KEY Insights

- ✓ Faster if the solver supports unsat core extraction and assumptions
 - ❑ Most solvers provide an assumption interface
 - ❑ Does not require many modifications
- ✓ Depends on the ordering of the clauses

*Enumerate all Minimal Unsatisfiable Subsets and
Minimum Correction Subsets*

MUS extraction

[cpmpy/examples/tutorial_ijcai22/2_musx_ocus.ipynb](#)

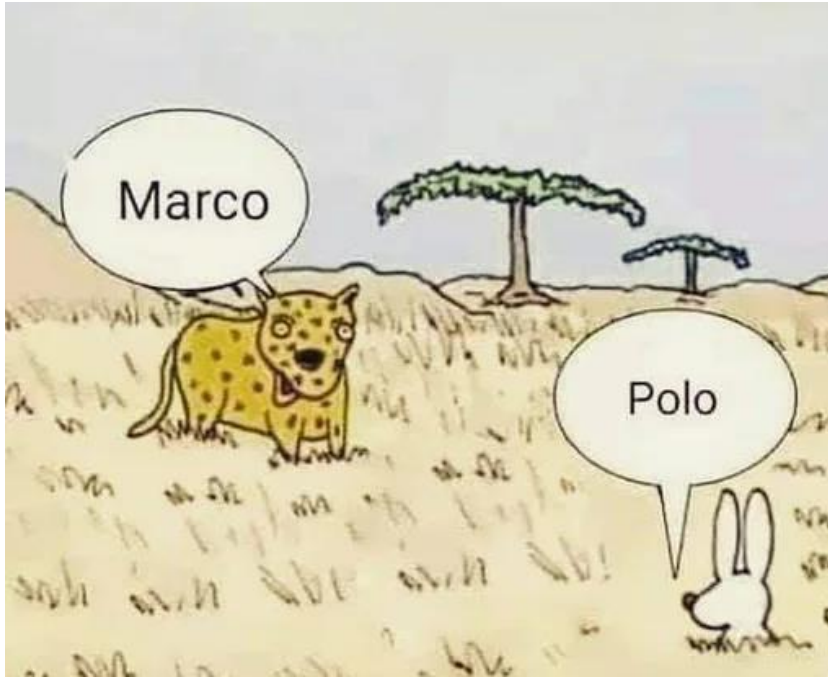
Master \leftrightarrow sub-problem



Marco Polo
(1254 –1324)

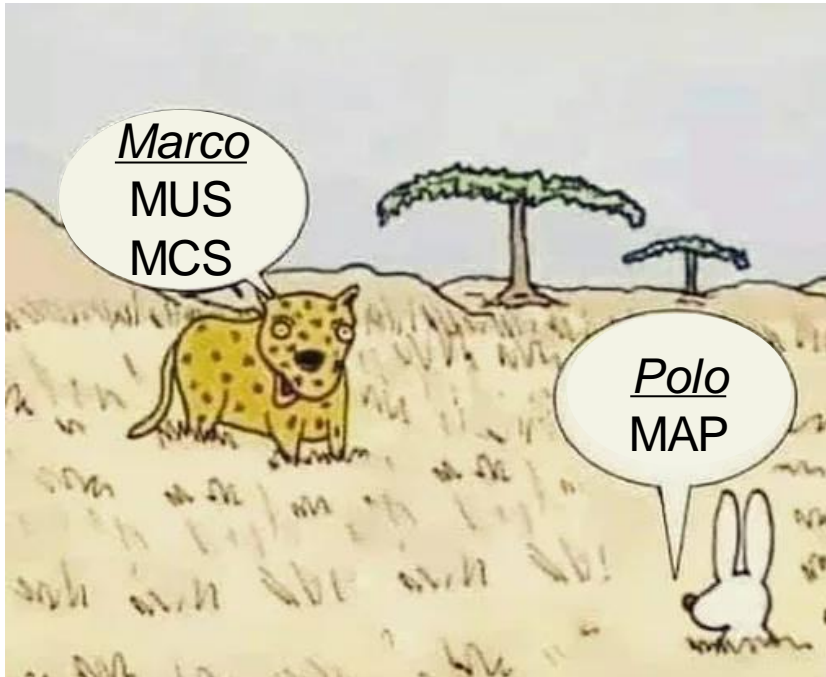
Marco-Polo

An example of Master \leftrightarrow Sub-problem approach



Marco-Polo

An example of Master \leftrightarrow Sub-problem approach



MARCO: Mapping Regions of
Constraints sets

- MUS enumeration algorithm

POLO: Power Logic

- “Map” of the powerset as a propositional logic Formula

MARCO Algorithm

Input: Constraint system C

$Map \leftarrow T$

while Map is satisfiable:

$seed \leftarrow \text{getUnexplored}(Map)$

if $seed$ is satisfiable:

$MSS \leftarrow \text{grow}(seed, C)$

output MSS

$Map \leftarrow Map \wedge \text{blockDown}(MSS)$

else:

$MUS \leftarrow \text{shrink}(seed, C)$

output MUS

$Map \leftarrow Map \wedge \text{blockUp}(MUS)$



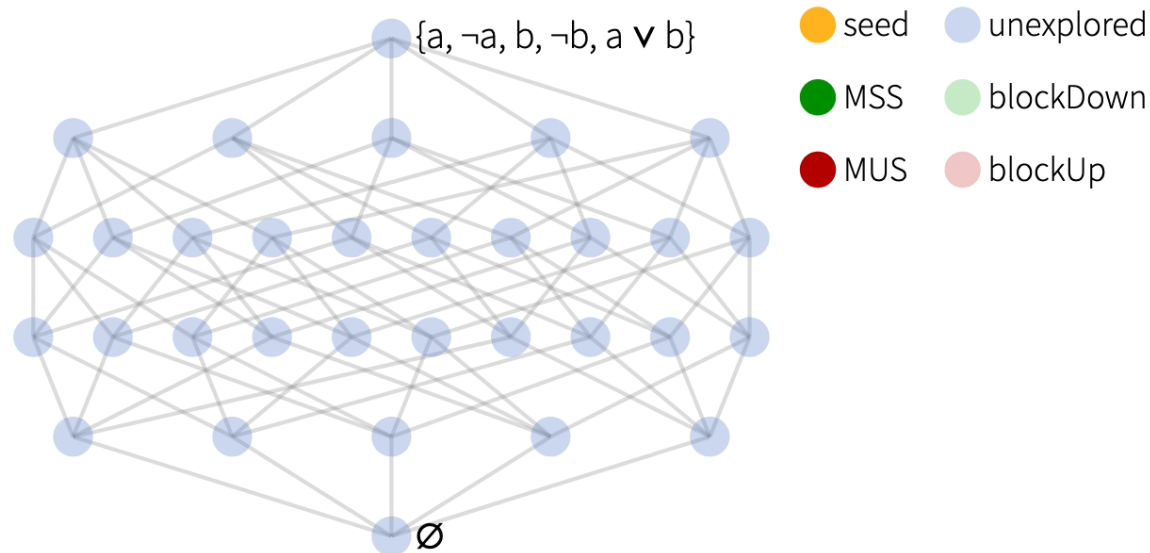
Example input: $C = \{a, \neg a, b, \neg b, a \vee b\}$

$seed$:

Map :

$MUSes$:

$MSSes$:



Demo

MarcoPolo MUS/MCS enumeration

[cpmpy/examples/tutorial_ijcai22/marco-mus-mcs-enumeration.ipynb](#)

Explaining UNSAT with MUSes

Methods & Insights

- Deletion-based MUS extracts only 1MUS
 - Efficiently enumerate all Minimal Unsatisfiable Subsets and Minimum Correction Subsets
 - No guarantee of **subset-minimality** (only heuristically)
 - **Smallest** Minimal Unsatisfiable Subset (SMUS)
- or **optimality** with weighted constraints
- **Optimal (Constrained)** Unsatisfiable Subset (OCUS)

Master \leftrightarrow Sub-problem

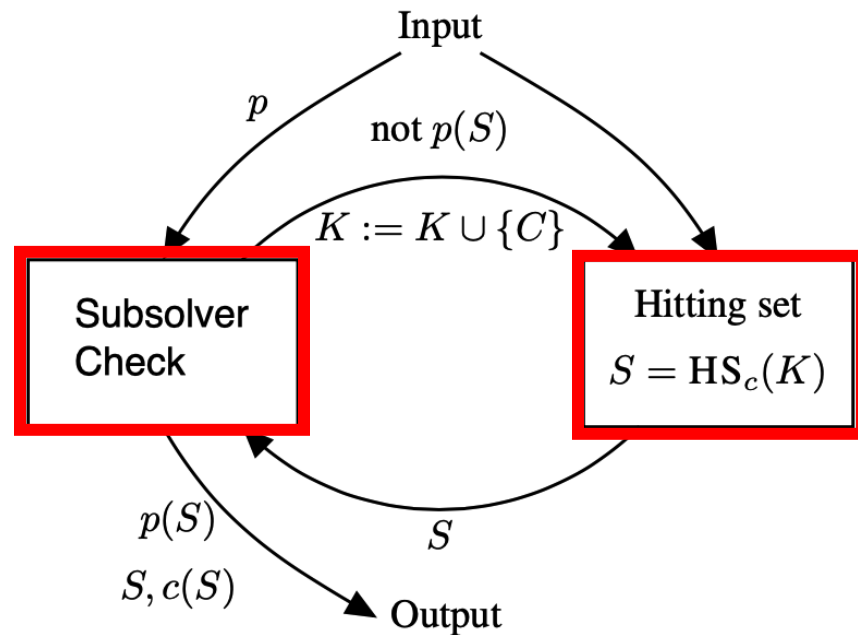
Implicit hitting set algorithms

Master \leftrightarrow Sub-problem approach

Implicit hitting set algorithms

General Structure:

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



Master \leftrightarrow Sub-problem approach

Implicit hitting set algorithms

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

- Deciding whether a MUS of size $\leq k$ is Σ_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 $S \subseteq F$ is a MCS of F if and only if it is a *minimum hitting set* of $MUSs(F)$.

A set $S \subseteq F$ is a MUS of F if and only if it is a *minimum hitting set* of $MUSs(F)$.

[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

Let

\mathcal{F} be an unsatisfiable formula,

$f: 2^{\mathcal{F}} \rightarrow \mathbb{N}$ a *cost function*

$p: 2^{\mathcal{F}} \rightarrow \{t, f\}$ a *predicate (constraint)*

We call $\mathcal{S} \subseteq \mathcal{F}$ an OCUS of \mathcal{F} (with respect to f and p) if

- \mathcal{S} is unsatisfiable,
- $p(\mathcal{S})$ is true
- all other unsatisfiable $\mathcal{S}' \subseteq \mathcal{F}$ with $p(\mathcal{S}') = t$ satisfy $f(\mathcal{S}') \geq f(\mathcal{S})$.

Optimal Constrained Unsatisfiable Subsets

Implicit hitting set-based algorithm

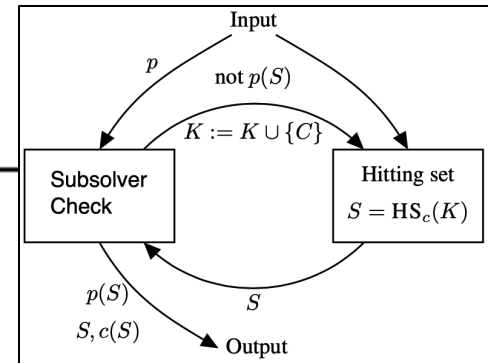
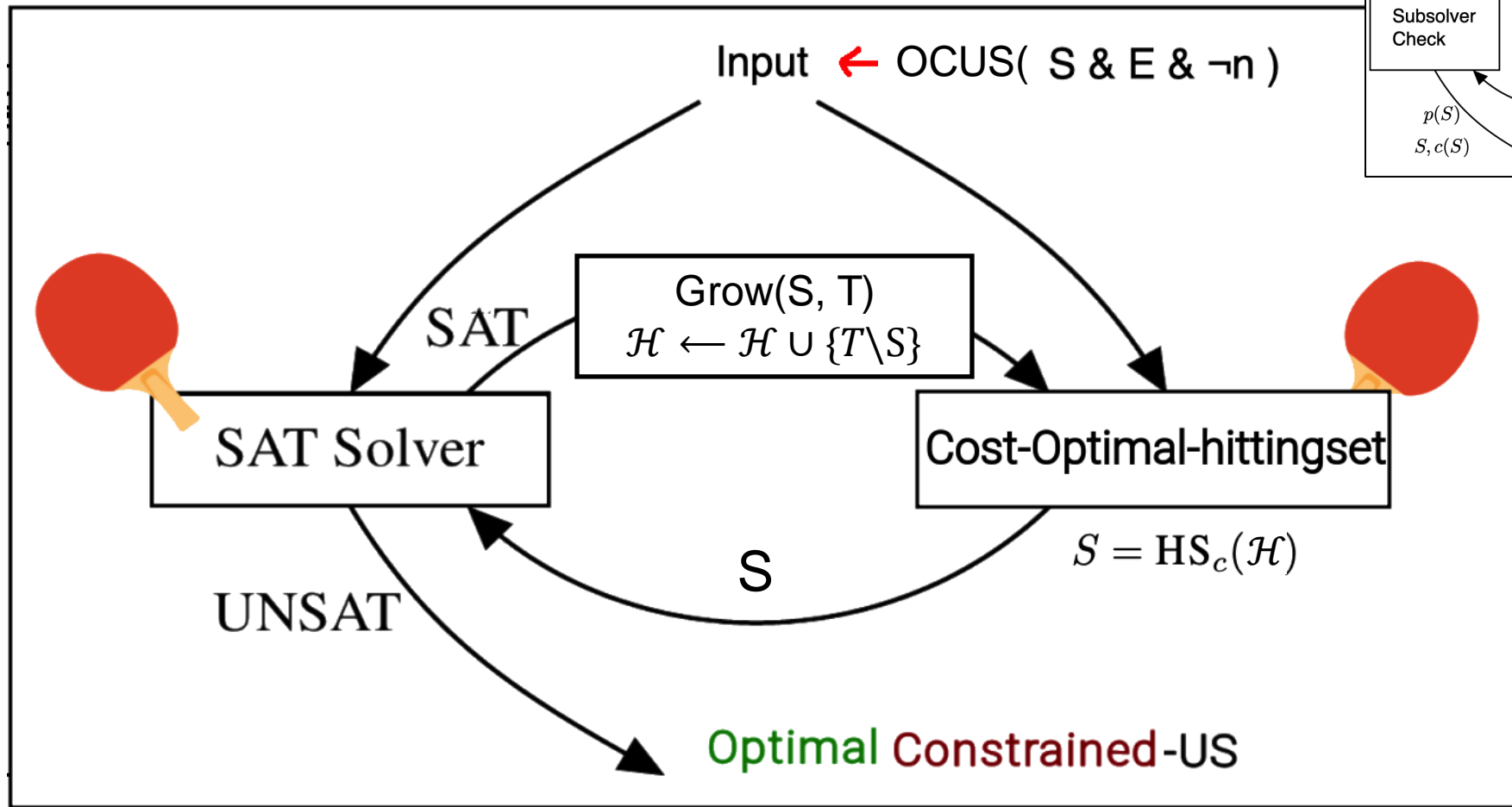
→ uses an implicit hitting set algorithm (like SMUS and MaxHS)

Algorithm 4: OCUS(T, f, p)

```
1  $\mathcal{H} \leftarrow \emptyset$ 
2 while true do
3    $\mathcal{S} \leftarrow \text{COST-OPTIMAL-HITTINGSET}(\mathcal{H}, f, p)$ 
4   if  $\neg \text{SAT}(\mathcal{S})$  then
5     return  $\mathcal{S}$ 
6   end
7    $\mathcal{S} \leftarrow \text{GROW}(\mathcal{S}, T)$ 
8    $\mathcal{H} \leftarrow \mathcal{H} \cup \{T \setminus \mathcal{S}\}$ 
9 end
```

→ **MIP solver**

→ **SAT/CP solver as an oracle**



OCUS with assumptions

```
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)
```

Hitting set Solver

```
while(True):  
    hittingset_solver.solve()  
    # Get hitting set  
    hs = ind[ind.value() == 1]  
  
    if not assum_solver.solve(assumptions=hs):  
        return soft[ind.value() == 1]  
  
    # compute complement of model in formula F  
    C = ind[ind.value() != 1]  
  
    # Add complement as a new set to hit: sum x[j] * hij >= 1  
    hittingset_solver += (sum(C) >= 1)
```

repeatedly
compute
hitting sets

CP/SAT
as an oracle

Extract
Correction Subset

MUS extraction

[cpmpy/examples/tutorial_ijcai22/2_musx_ocus.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 expands.
 - Big efficiency gains if incremental (and not restart)!
 - 2. CP/SAT is used as an oracle**
 - CP/SAT checking satisfiability of a subset
 - Grow solves a MaxSAT problem, s.t. complement is a (small) MCS

Part 3: Advanced examples of
Master \leftrightarrow Sub-problem solving
*Implicit Hitting set and Cutting plan
algorithms*

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?

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

What if a model is SAT?

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

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

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

Explanation step

Let E' & $S' \Rightarrow n$ be one explanation step.

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

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 $E' \& S' \Rightarrow n$
(Sudoku) All different column, row, box constraints

n = a newly derived fact

How? $MUS(\neg n \& E \& S)$ 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
```

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

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

$sol\text{-}to\text{-}explain = \text{propagate}(E \ \& \ S) \setminus E$

$p = \text{exactly-one}(\{\sim n \mid n \in sol\text{-}to\text{-}explain\}),$

return OCUS($n \mid n \in sol\text{-}to\text{-}explain$) & S & E & $\{\sim, f, p\}$

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

A sequence of explanations to explain SAT

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

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

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

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

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

Demo

OCUS for explaining SUDOKU

[cpmpy/examples/tutorial_ijcai22/4_explain_sudoku.ipynb](#)

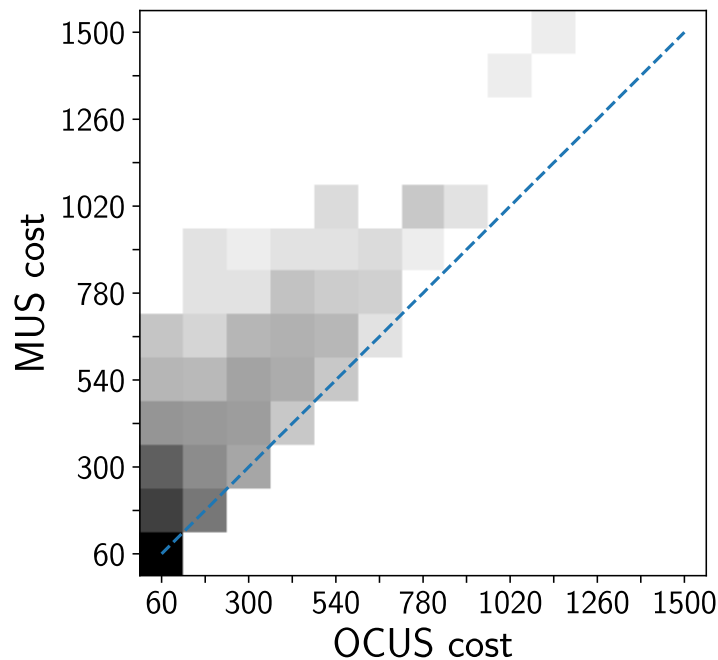
The best/easiest explanation step...

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
```

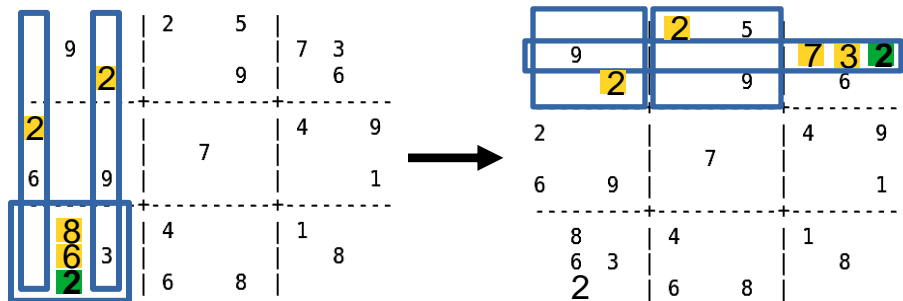


```
p = exactly-one({ ~n | n ∈ sol-to-explain }),
OCUS( { ~n | n ∈ sol-to-explain } & E & S, f, p)
```



Incrementality at the Sequence-level

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



\boxed{S} Model constraints

- Do not change from an explanation step to another

$\textcolor{yellow}{E}$ Derived facts of E

- Precision-increasing!

Incrementality at the Sequence-level

In practice

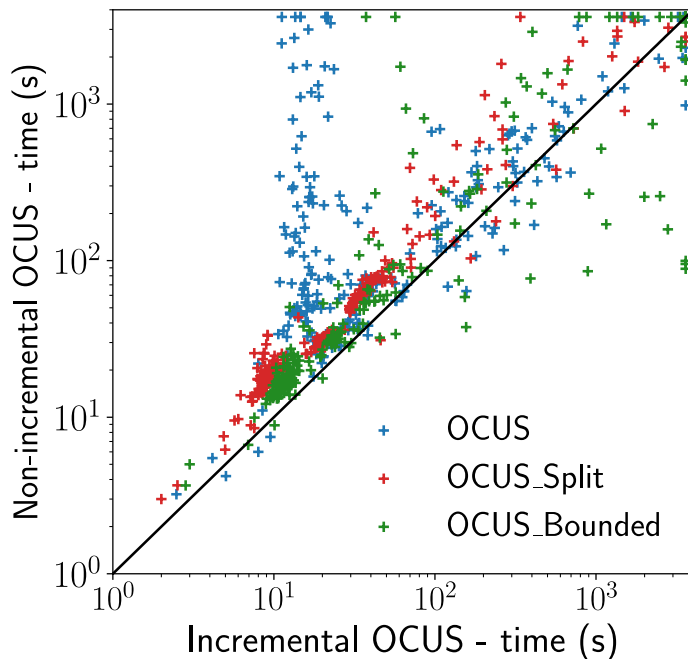
$$\text{OCUS}(\boxed{S} \ \& \ \boxed{E} \ \& \neg \ \boxed{n} \ , f, p)$$

Incremental OCUS works with the full unsatisfiable formula of step 0

$$\boxed{S} \ \& \ \boxed{E_{end}} \ \& \ \{\neg \ \boxed{n} \mid \boxed{n} \in \text{sol-to-explain}\}$$

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

- Underived facts cannot be taken
- Negated facts ($\neg \boxed{n}$) already explained should not 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