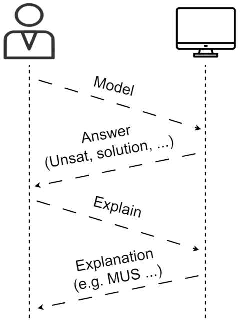
L03: Solving, debugging and explanation techniques



Prof. Tias Guns and Dr. Dimos Tsouros

KU LEUVEN

Partly based on slides from Pierre Flener, Uppsala University.

Outline

Model + **Solve**

Declarative problem solving: We model **what** – the solver takes care of the **how** …

We saw how to model a combinatorial problem in a CP modeling language...

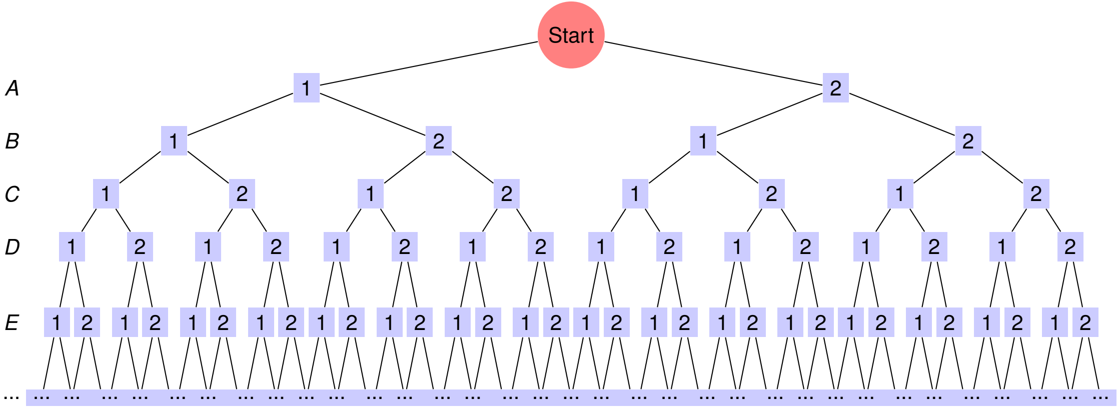
Now, we want to *solve* it!

Combinatorial problems:

- Huge search space
- Exponential growth of possible solutions
  - For n variables with d possible values each, the search space size is d^n
- Inference based on the constraints helps to prune infeasible solutions early, reducing the search space

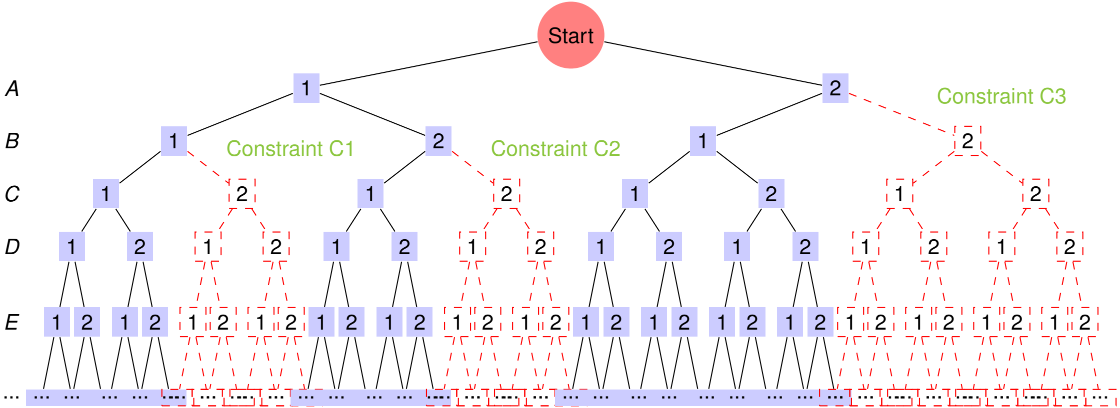
Solving combinatorial problems

Combinatorial problems: Huge search space



Solving combinatorial problems

Combinatorial problems: Huge search space, **need intelligent search!**



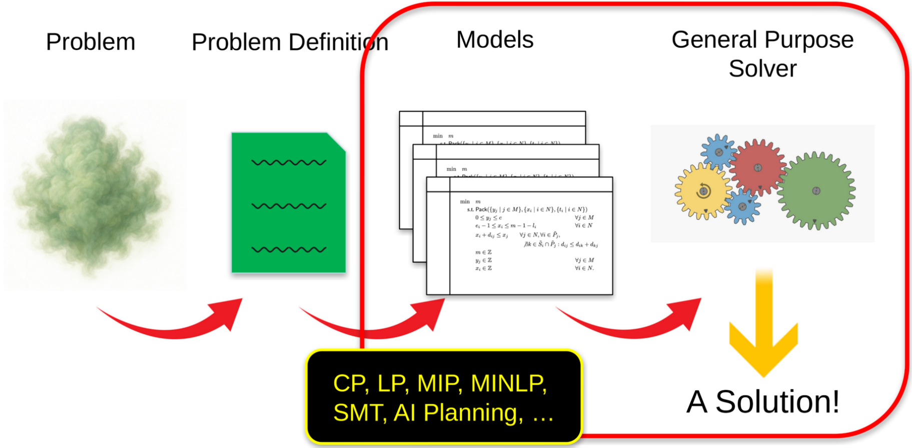
Solving combinatorial problems

Different solvers/solving technologies can be used for that. They differ in:

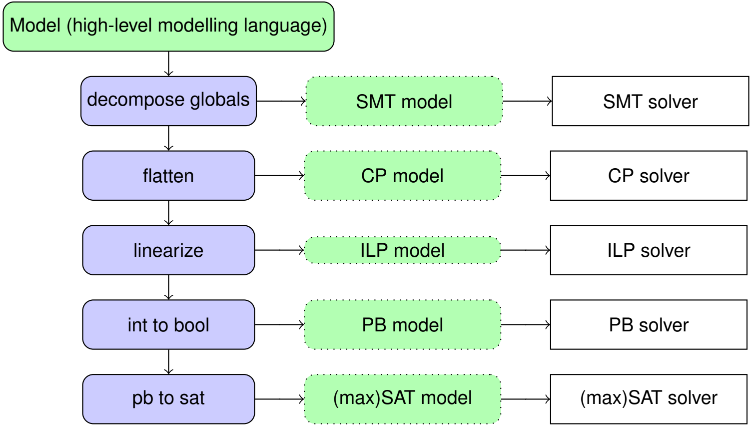
- The constraints they support (including global constraints/functions)
- How they perform search and propagation (CP vs MIP vs PB vs SAT)
- How they guide the search (heuristics, hyper-parameters)
- …

Encoding to solver-specific input

High-level CP modeling languages have to encode problems into a solver-specific input format.



Model Transformations



Exxample solvers:

- SMT: Z3
- CP: Or-Tools, Choco, GCS, Minizinc (modeling lang)
- ILP: Gurobi
- PB: Exact
- SAT: PySAT

Solving in CPMpy

Declarative modeling, easy solving

```
grid = cp.intvar(1,9, shape=(9,9), name="grid")  # Decision variables
model = cp.Model(
    [cp.AllDifferent(row) for row in grid],
    [cp.AllDifferent(col) for col in grid.T],  # numpy's Transpose
    [cp.AllDifferent(grid[i:i+3, j:j+3]) \
        for i in range(0, 9, 3) for j in range(0, 9, 3)]
)

# solve with default solver: ortools
model.solve()
```

Can also specify the solver to use:

```
model.solve("choco") # use choco solver - needs pychoco package
model.solve("gurobi") # use gurobi solver - needs gurobipy package
```

See what solvers you have in your machine:

```
cp.SolverLookup.solvernames()
```

Solving vs solution enumeration

Is one solution sufficient? In many problems no!

Finding all (or multiple) solutions by ’blocking’ each found solution:

```
while solutions_found < solution_limit:
  solve problem
  Add constraint that forbids the exact same solution
```

Solving vs solution enumeration – CPMpy

Is one solution sufficient? In many problems no!

Finding all (or multiple) solutions by ’blocking’ each found solution::

```
solutions = 0  # initialize
solution_limit = 5  # find 5 solutions

# model.solve() returns true if a solution is found
while model.solve() and solutions < solution_limit:
    solutions += 1
    # Constraint to enforce different solution
    model.add(~cp.all(grid == grid.value()))
```

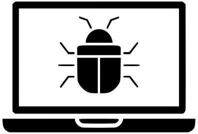
or just use `model.solveAll()` ← It returns the amount of solutions found, accessing the solutions:
https://cpmpy.readthedocs.io/en/latest/multiple_solutions.html

Outline

Debugging

You solve the problem, but
you get an **error**,
or no error, but also **no (correct) solution...**
Annoying, you have a **bug.**

How do you **debug** a model?



Debugging

General advise for debugging when modeling from expert modeller **Håkan Kjellerstrand**:

- Test the model **early and often.** This makes it easier to detect problems in the model.
- When a model is not working, **activate the constraints one by one** (e.g. comment out the other constraints) to test which constraint is the culprit:
  for each constraint *c* in *Constraints* **do**
      print "Trying:", *c*
      Solve c
- **Check the domains** (see lower). The domains should be as small as possible, but not smaller. If they are too large it can take a lot of time to get a solution. If they are too small, then there will be no solution.

Debugging – CPMpy

General advise for debugging when modeling from expert modeller **Håkan Kjellerstrand**:

- Test the model **early and often.** This makes it easier to detect problems in the model.
- When a model is not working, **activate the constraints one by one** (e.g. comment out the other constraints) to test which constraint is the culprit.

```
for c in model.constraints:
    print("Trying",c)
    cp.Model(c).solve()
```

- **Check the domains** (see lower). The domains should be as small as possible, but not smaller. If they are too large it can take a lot of time to get a solution. If they are too small, then there will be no solution.

Debugging

The bug can be situated in one of three layers:

1. your model
2. the modeling library (CPMpy)
3. the solver

Ordered from most likely to least likely!

Bug in the solver

You try with the default solver (or another one) and you get an error, or not the desired solution.

Use a different solver and observe:

1. Outcome changes! It was a (rare) solver bug. Report it to the bug tracker of the modeling library or directly to the solver developers!
2. Outcome is the same! Not a solver error after all . . .

Debugging a modeling error – CPMpy

You get an error when you create an expression?
Quirks in Python/CPMpy (from last lecture):

1. **Logical and/or**: Use & and |, and make sure to always put the subexpressions in brackets.

Example

write `(x == 1) & (y == 0)` instead of `x == 1 & y == 0`. The latter won't work. Python will think you meant `x == (1 & y) == 0`.

2. you can write `vars_list[other_var]` but you can't write `non_var_list[a_var]`. That is because the vars list knows CPMpy, and the `non_var_list` does not. Wrap it: `non_var_list = cp.cpm_array(non_var_list)` first.
3. CPMpy overloads all/any/max/min/sum/abs to create expressions with them. Always use `cp.sum(v)` instead of `sum(v)`. You can also use directly NumPy's `v.sum()` instead, if `v` is a matrix or tensor.

Debugging a modeling error – CPMpy

You get an error when you create an expression . . . But you do not know why!

Print the constraints you create (or the subexpressions), and check that the output matches what you wish to express!

Example

The following:

```python
x = cp.intvar(0,5,shape=(2,2))
con = sum(x)
print(con)
```

will print [(IV0) + (IV2) (IV1) + (IV3)] and you can see that it is not really a sum, but a list!

Solution: Use `cp.sum(x)` instead!

Outline

Explainable Constraint Solving

You model the problem and you solve! No Error!

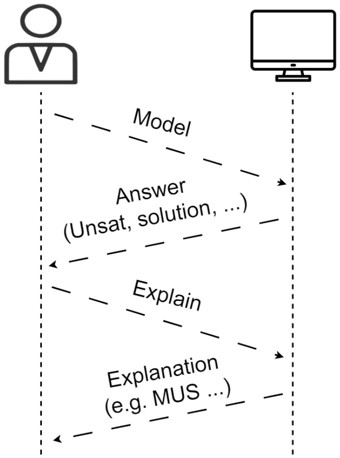
But also:

- What if the model is UNSAT?
- What if the solution is unexpected?
- What if the solution is not good enough?

There is a modeling error . . . Or the problem constraints are too tight . . .

Explainable AI: Human-Aware AI systems that interact with the users to assist in decision making

Mode of interaction



Explainable Constraint Solving

In general, ”Why X?” (with X (part of) a solution or UNSAT)
2 patterns of explanations:

1. **Deductive explanation**: How was X derived? (Why I didn’t get any solution?)
2. **Counterfactual explanation**: Why X and not Z? (How can I make it satisfiable?)

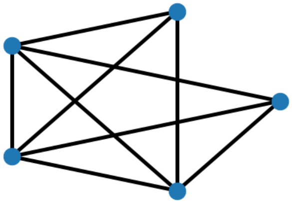
Running Example

Example (Graph Colouring)

Graph colouring is the problem of assigning colours to the nodes of a graph, such that no two **adjacent** nodes share the same colour.

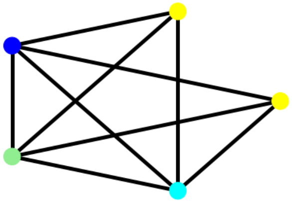
- Variables are the nodes, possible values are the colours:
  $\text{node}_i \in \{1, 2, \ldots, \text{max\_colors}\}, \quad \forall i \in \text{Nodes}$
- Constrain edges to have differently colored nodes (i.e., not equal values):
  $\text{node}_1 \neq \text{node}_2, \quad \forall(\text{node}_1, \text{node}_2) \in \text{Edges}$

Initial graph:



$\longrightarrow$

Coloured graph:



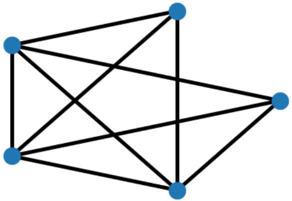
Running Example – CPMpy

Example (Graph Colouring)

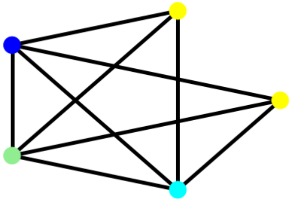
Graph colouring is the problem of assigning colours to the nodes of a graph, such that no two **adjacent** nodes share the same colour.

```
m = cp.Model()
# variables are the nodes, possible values are the colours
nodes = cp.intvar(1, max_colors, shape=nodes_num, name="Node")
# constrain edges to have differently colored nodes (i.e., not equal values)
m.add([nodes[n1] != nodes[n2] for n1, n2 in graph.edges()])
```

Initial graph:



Coloured graph:



Outline

Graph Colouring: Unsatisfiable

But what if our problem is not satisfiable?

- e.g. we have less colours available than needed!

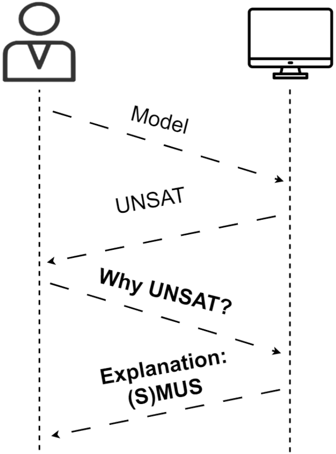
Example

```
m, nodes = graph_coloring(G, max_colors=3)
No solution found.
```

Explanation techniques can help us understand:

- Why is it unsatisfiable? (Deductive explanation)
- How to fix it? (Counterfactual explanation)

Deductive Explanations



- Find the cause!
- Why X? (e.g. why is it UNSAT?)

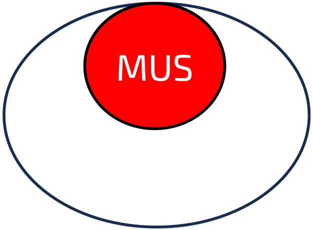
Deductive Explanations

Question: "Why is it unsatisfiable?"

- Answer: "The set of all constraints cannot be satisfied."

Not very useful ...

- Answer: "This (small) subset of constraints cannot be satisfied together!"

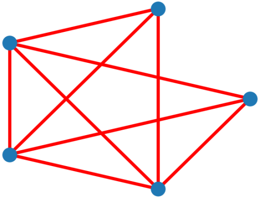


Pinpoint to a subset of constraints causing a conflict ...

Deductive Explanations: Graph colouring

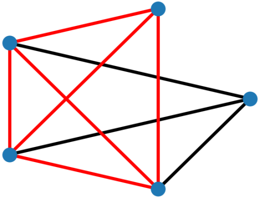
Question: ”Why is my graph colouring problem unsatisfiable?”

- Answer: ”I cannot colour this graph with all these constraints.”



Not very useful ...

- Answer: ”These constraints prevent me from finding a solution!”

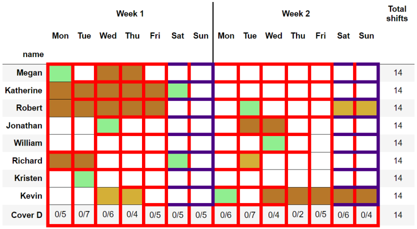


Pinpoint to the subset of constraints causing a conflict ...

Deductive Explanations: Nurse Rostering

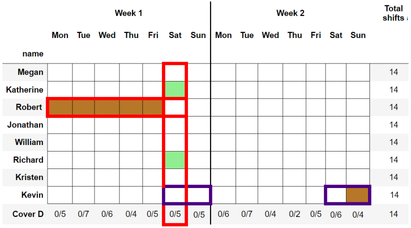
Question: ”Why is my nurse rostering problem unsatisfiable?”

- Answer: ”I cannot schedule satisfying all these constraints.”



Not very useful ...

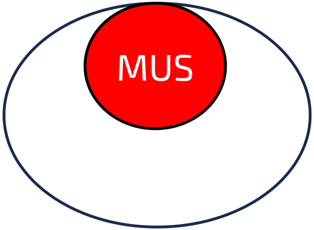
- Answer: ”These constraints prevent me from finding a solution!”



Pinpoint to the subset of constraints causing a conflict ...

Minimal Unsatisfiable Subset (MUS)

The cause of UNSAT: A set of constraints that cannot be satisfied in conjunction!



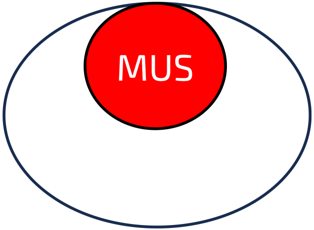
Definition (Minimal Unsatisfiable subset (MUS))

A subset of constraints $C' \subseteq C$ is called a MUS of C if:

- C' is unsatisfiable, i.e., $solve(C') = UNSAT$.
- C' is minimal, i.e., $\forall c \in C'$, $solve(C' \setminus \{c\}) = SAT$.

Minimal Unsatisfiable Subset (MUS)

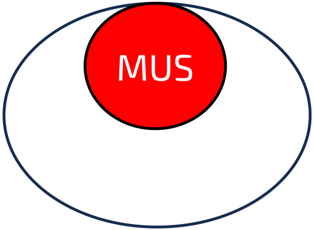
The cause of UNSAT: A set of constraints that cannot be satisfied in conjunction!



- Explain the cause.
- Pinpoint to constraints causing a conflict.
- Trim model to a minimal set of constraints.
- Minimize cognitive burden for user.

Minimal Unsatisfiable Subset (MUS)

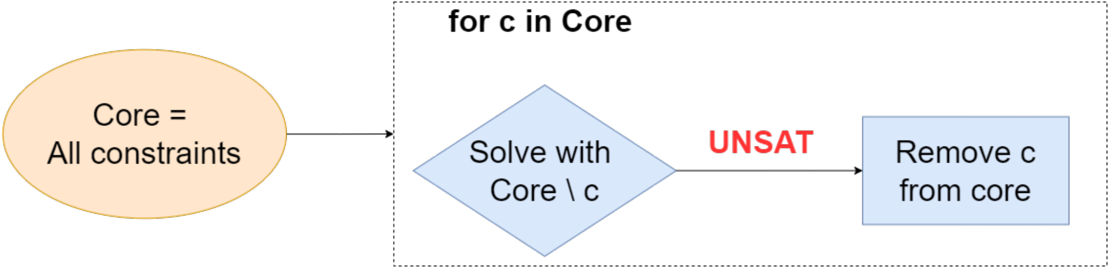
A cause of UNSAT: A set of constraints that cannot be satisfied in conjunction!



- Explain a cause. Important! Explain one of the (possibly many) causes
- Pinpoint to constraints causing a conflict.
- Trim model to a minimal set of constraints.
- Minimize cognitive burden for user.

Computing MUSes

Multiple ways to compute MUSes. Deletion-based MUS algorithm:



Computing MUSes – CPMpy

Multiple ways to compute MUSes. Deletion-based MUS algorithm:

Example (Deletion-based MUS algorithm)

```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
```

Computing MUSes

Deletion-Based MUS - Example:

1	2	3	4	5	6	7	8	UNSAT	
1	2	3	4	5	6	7	8	UNSAT	
1	2	3	4	5	6	7	8	UNSAT	
1	2	3	4	5	6	7	8	UNSAT	
1	2	3	4	5	6	7	8	SAT	Keep c4!
1	2	3	**4**	5	6	7	8	UNSAT	
1	2	3	**4**	5	6	7	8	SAT	Keep c6!
1	2	3	**4**	5	**6**	7	8	SAT	Keep c7!
1	2	3	**4**	5	**6**	**7**	8	UNSAT	

Check

Removed

In MUS

MUS: **{c4,c6,c7}**

Computing MUSes

- Simple deletion-based approach is the baseline
- Use Assumption-based solving.
  - Extract UNSAT core from solver ... and exploit incremental solving!
- Divide-and-conquer approach $\rightarrow$ QuickXplain.
  - Binary search: remove half the constraints for each check

Computing MUSes – CPMpy

- Simple deletion-based approach is the baseline

Example

```
from cpmpy.tools.explain.mus import mus_naive
```

- Use Assumption-based solving.
  - Extract UNSAT core from solver ... and exploit incremental solving!

Example

```
from cpmpy.tools.explain.mus import mus
```

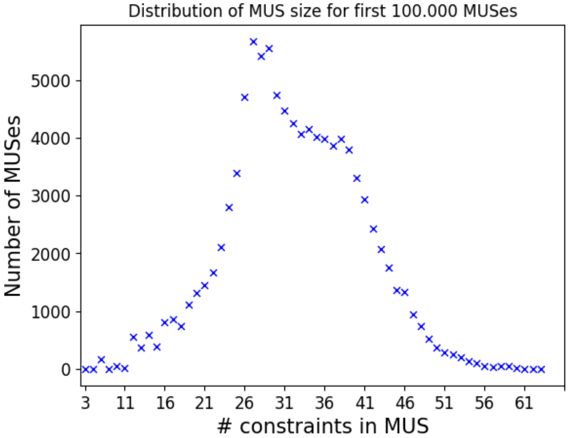
- Divide-and-conquer approach $\rightarrow$ QuickXplain.
  - Binary search: remove half the constraints for each check

Example

```
from cpmpy.tools.explain.mus import quickxplain
```

Which MUS to return?

Multiple MUSes may exist!
The nurse rostering problem we saw has 100k+ MUSes!



Which MUS to return?

Multiple MUSes may exist!
The nurse rostering problem we saw has 100k+ MUSes!

- Which one to show?
- Smaller MUSes may be more **understandable.**
- Some MUSes may involve more **understandable** constraints than others.
- Can we influence which MUS to find and show?

Which MUS to return?

Definition (Optimal Unsatisfiable Subset (OUS))

Given a set of constraints C, with each constraint associated with a weight, an OUS is a MUS $C' \subseteq C$ that minimizes the sum of weights: $min \sum_{c_i \in C'} w_i \cdot c_i$.

Based on the fact that some constraints may be easier to understand than others!

Definition (Smallest Unsatisfiable Subset (sMUS))

Given a set of constraints C, an sMUS is a Minimal Unsatisfiable Subset $C' \subseteq C$ that minimizes the cardinality: *minimize* $|C'|$. An sMUS is an OUS in the case that all constraints have equal weights.

Typically, in explanations also smaller is better: Explaining with the fewest constraints is possibly good enough!

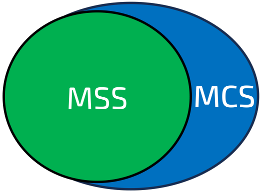
Key concepts used for finding optimal MUSes

Definition (Maximal Satisfiable Subset (MSS))

Given a set of constraints C, an MSS is a subset $C' \subseteq C$ that is satisfiable and maximal, meaning there is no constraint $c \in C \setminus C'$ such that $C' \cup \{c\}$ remains satisfiable.

Definition (Minimal Correction Subset (MCS))

Given a set of constraints C, an MCS is a subset $C' \subseteq C$ that is minimal such that $C \setminus C'$ is satisfiable. In other words, an MCS is a smallest subset of constraints whose removal results in a Maximal Satisfiable Subset (MSS).



Key concepts used for finding optimal MUSes

Definition (Hitting Set)

Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, a hitting set is a subset $H \subseteq \bigcup_{i=1}^{n} S_i$ such that $H \cap S_i \neq \emptyset$ for every $S_i \in \mathcal{S}$. In other words, a hitting set contains at least one element from each set in the collection.

Definition (Hitting set duality)

A MUS is a hitting set of all MCSes, and an MCS is a hitting set of all MUSes. Let M be the collection of all MUSes, and S be the collection of all MCSes.

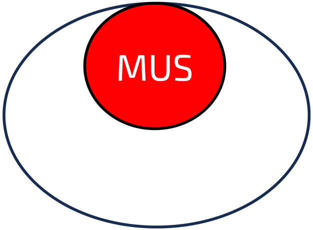
- Every MUS $M_i \in M$ is a hitting set of all MCSes S: $\forall S_j \in S$, $M_i \cap S_j \neq \emptyset$.
- Every MCS $S_j \in S$ is a hitting set of all MUSes M: $\forall M_i \in M$, $S_j \cap M_i \neq \emptyset$.

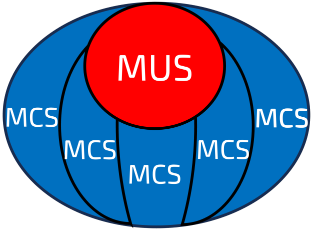
Key concepts used for finding optimal MUSes

Definition (Hitting set duality)

A MUS is a hitting set of all MCSes, and an MCS is a hitting set of all MUSes.
Let M be the collection of all MUSes, and S be the collection of all MCSes.

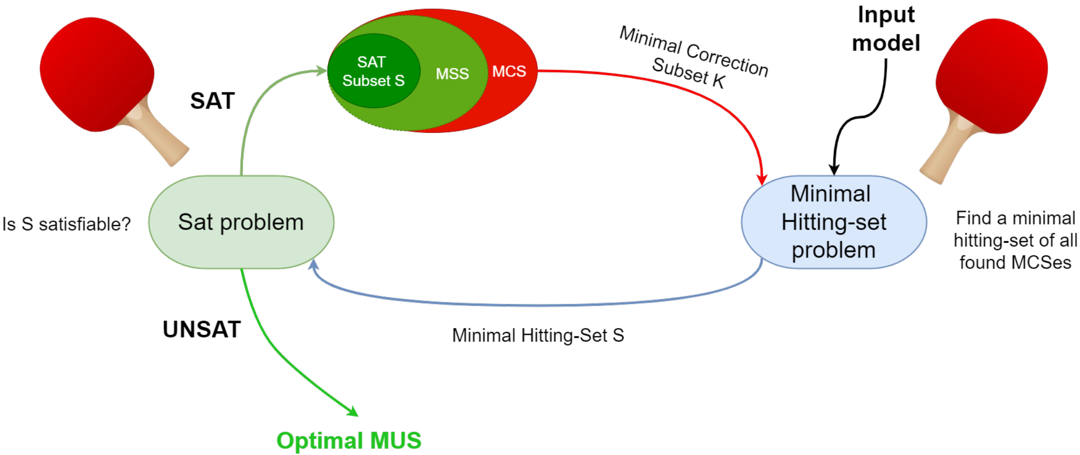
- Every MUS $M_i \in M$ is a hitting set of all MCSes S: $\forall S_j \in S$, $M_i \cap S_j \neq \emptyset$.
- Every MCS $S_j \in S$ is a hitting set of all MUSes M: $\forall M_i \in M$, $S_j \cap M_i \neq \emptyset$.





Optimizing which MUS is found

Find all MUSes, and pick the best? NO! Potentially exponential number of MUSes



Optimizing which MUS is found

Find all MUSes, and pick the best? NO! Potentially exponential number of MUSes

Algorithm: $\textsc{ocus}(\mathcal{F}, f, p)$

$\mathcal{H} \leftarrow \emptyset$ // Collection of sets-to-hit
while true **do**
  $\mathcal{S} \leftarrow \textsc{cost-optimal-hittingset}(\mathcal{H}, f, p)$ // hitting set
  if $\neg\mathrm{SAT}(\mathcal{S})$ **then**
    return $\mathcal{S}$ // OCUS found!
  end
  $\mathcal{S} \leftarrow \textsc{Grow}(\mathcal{S}, \mathcal{F})$ // Grow Satisfiable subset
  $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathcal{F} \setminus \mathcal{S}\}$ // Add correction subset (new set-to-hit)
end

Finding optimal MUSes – CPMpy

OCUS algorithm for finding the optimal MUS:

Example

```
from cpmpy.tools.explain.mus import optimal_mus
optimal_mus(constraints, weights=...)
```

OCUS algorithm for finding the smallest MUS (default weights are equal):

Example

```
from cpmpy.tools.explain.mus import optimal_mus
optimal_mus(constraints)
```

Can directly use `smus`, which uses `optimal_mus` as above:

Example

```
from cpmpy.tools.explain.mus import smus
smus(constraints)
```

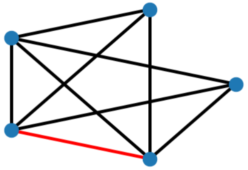
Counterfactual explanations

Not always enough to explain the cause!

How to **change the model**, in order to find a solution?

- Find constraints that, if *removed*, a solution can be found!
- Find a correction subset . . .

”Removing this constraint will make our problem satisfiable”



Reminder:

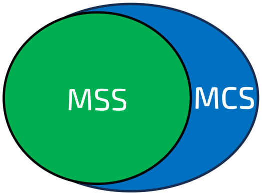
Definition (Minimal Correction Subset (MCS))

Given a set of constraints C, an MCS is a subset $C' \subseteq C$ that is minimal such that $C \setminus C'$ is satisfiable. In other words, an MCS is a smallest subset of constraints whose removal results in a Maximal Satisfiable Subset (MSS).

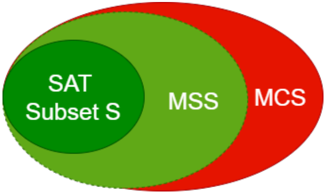
Computing MCSes

MCSes can be used to provide counterfactual explanations . . . How to compute them?

Key property:
MCSes are the complement of MSSes!!



Grow a set of constraints $C' \subseteq C$ until UNSAT! **Take the complement.**



Computing MCSes – CPMpy

Simple growing-based approach (similar to deletion-based MUS):

Example (Grow-based MCS/MSS)

```
def mcs_naive(constraints):
    mss = []  # grow a satisfiable subset one-by-one
    mcs = []  # everything else is in the minimum conflict set

    for cons in constraints:
        if cp.Model(mss + [cons]).solve():
            mss.append(cons)  # adding it remains SAT
        else:
            mcs.append(cons)  # UNSAT, causes conflict

    return mcs
```

→ Finds any MSS/any MCS!

→ Many may exist!

Computing optimal MCSes

Maximize the number of satisfied constraints ... treat it as (weighted) MAX-CSP!
$\rightarrow$ One *optimization* problem, instead of multiple *satisfaction* ones ...
$\rightarrow$ MAX-CSP: Find a solution that satisfies a maximum number of constraints.
$\rightarrow$ Finds largest MSS = complement of smallest MCS!

- (Half-)Reify all constraints in C, creating boolean indicator variables for each:

$$b_i \rightarrow c_i, \forall c_i \in C$$

- and maximize the sum of the values of reification variables:

$$\text{maximize} \sum_{i=1}^{|C|} b_i$$

- Simply take the constraints not satisfied:

$$MCS = \{c_i \in C \mid \neg b_i\}$$

Computing optimal MCSes – CPMpy

Maximize the number of satisfied constraints ... treat it as (weighted) MAX-CSP!
→ One *optimization* problem, instead of multiple *satisfaction* ones ...
→ MAX-CSP: Find a solution that satisfies a maximum number of constraints.
→ Finds largest MSS = complement of smallest MCS!

- (Half-)Reify all constraints in C, creating boolean indicator variables for each:

```
maxcsp_model = cp.Model()
B = cp.boolvar(shape=len(constraints))  # Boolean indicator variable for
each constraint
maxcsp_model.add(B.implies(constraints))  # reify constraints (vectorized)
```

- and maximize the sum of the values of reification variables:

```
maxcsp_model.maximize(cp.sum(B))  # maximize satisfied constraints
maxcsp_model.solve()
```

- Simply take the constraints not satisfied:

```
mcs = [c for b,c in zip(B, constraints) if b.value() is False]
```

Computing optimal MCSes

Maximize the number of satisfied constraints . . . treat it as (weighted) MAX-CSP!
$\rightarrow$ One *optimization* problem, instead of multiple *satisfaction* ones . . .
$\rightarrow$ MAX-CSP: Find a solution that satisfies a maximum number of constraints.
$\rightarrow$ Finds largest MSS = complement of smallest MCS!

Alternative:

- Let CPMpy handle the reification
- Use directly the (soft) constraints

```
maxcsp_model.maximize(cp.sum(constraints))  # maximize satisfied constraints
maxcsp_model.solve()
mcs = [c for c in constraints if c.value() is False]
```

Outline

Explaining solutions

"Why X?": Why is X part of the solution?

Explaining logical consequences:

Definition (Logical consequence)

Logical consequence: a variable assignment entailed by the constraints (and possible an initial partial assignment)

Is X a logical consequence? Try to solve the problem, enforcing $\neg X$:

- If SAT: no explanation, return new solution.
- If UNSAT: use any technique for explaining this UNSAT problem (MUS, MCS, ...).

Explaining optimality

"Why X?": Why this solution is optimal w.r.t. the objective function $f(x)$?

Explaining logical consequences!

- Taking into account also the objective value found!

Try to solve the problem, enforcing $f(x) < o$ (assume minimization problem), with o being the objective value of the optimal solution:

- Will be UNSAT, as we know o is optimal!
- Use any technique for explaining this UNSAT problem(MUS, MCS, ...).

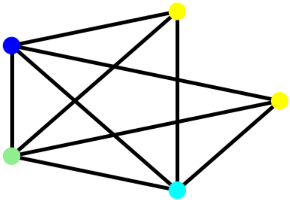
Explaining optimality

"Why X?": Why is this solution optimal w.r.t. the objective function $f(x)$?
Graph colouring is actually an optimization problem!

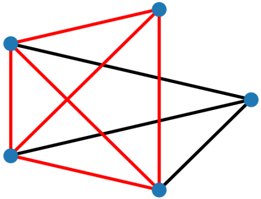
Example

```
# variables are the nodes, possible values are the colours
nodes = cp.intvar(1, max_colors, shape=nodes_num, name="Node")
# constrain edges to have differently colored nodes (i.e., not equal values)
m.add([nodes[n1] != nodes[n2] for n1, n2 in graph.edges()])
m.minimize(cp.max(nodes))  # minimize colours used!
```

Why does the best solution need 4 colours?



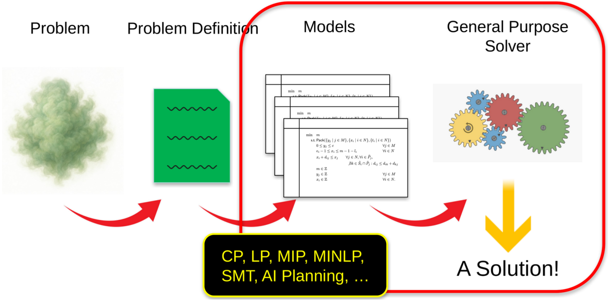
Because of these constraints!



Outline

Summary

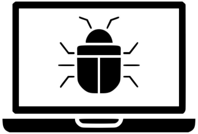
- Model + **Solve**



- Debugging a model
- Explainable Constraint Solving

Summary

- Model + **Solve**
- Debugging a model

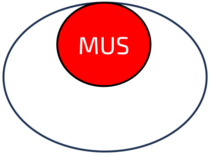


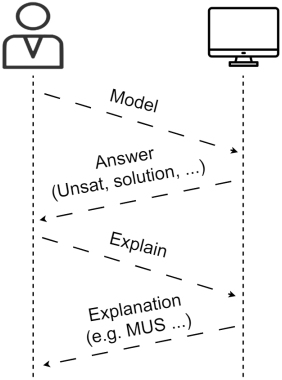
https://cpmpy.readthedocs.io/en/latest/how_to_debug.html

- Explainable Constraint Solving

Summary

- Model + **Solve**
- Debugging a model
- Explainable Constraint Solving





Advanced tutorial:
https://github.com/CPMpy/XCP-explain