L02: Basic Modeling

st, du, vi, tk, gw, kl :: {0,1}

52*st + 85*du + 60*vi + 84*tk + 117*kl + 35*gw <= 4*52

maximize(50*st + 80*du + 75*vi + 82*tk + 95*kl + 7*gw)

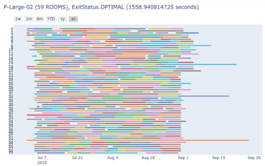
Prof. Tias Guns and Dr. Dimos Tsouros

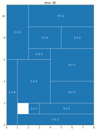
KU LEUVEN

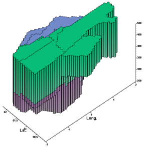
Partly based on slides from Pierre Flener, Uppsala University.

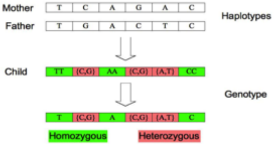
Outline

Combinatorial Optimisation

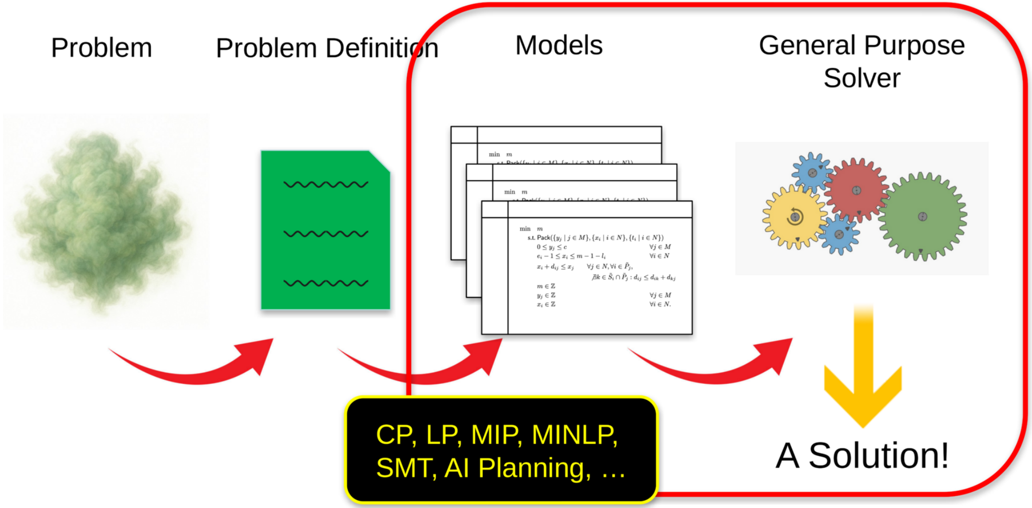








Model-and-Solve



Modelling (declarative) vs Programming (imperative)

Problem
(multi-eyed beast)

Problem Definition
(free form, unambiguous)

what?

how?

Model
(formal specification)

Algorithm
(pseudo-code)

automatic!

manual!

program

program

Example (Problem Definition: Sudoku)

The goal of Sudoku is to *complete* a partially filled 9x9 grid with numbers so that each row, column and 3x3 section contains each digit between 1 and 9 once.

Example (Model: Sudoku)

$$
\begin{array}{ll}
G_{ij} \in \{1, 2, \ldots, 9\} & \forall i, j \in \{1, \ldots, 9\} \\
\textsc{AllDifferent}(\{G_{ij} | j \in \{1, \ldots, 9\}\}) & \forall i \in \{1, \ldots, 9\} \\
\textsc{AllDifferent}(\{G_{ij} | i \in \{1, \ldots, 9\}\}) & \forall j \in \{1, \ldots, 9\} \\
\textsc{AllDifferent}(\{G_{kl} | k \in \{i, \ldots, i+2\}, l \in \{j, \ldots, j+2\}\}) & \forall i, j \in \{1, 4, 7\} \\
G_{ij} = v & \forall (i, j, v) \in \mathcal{D}
\end{array}
$$

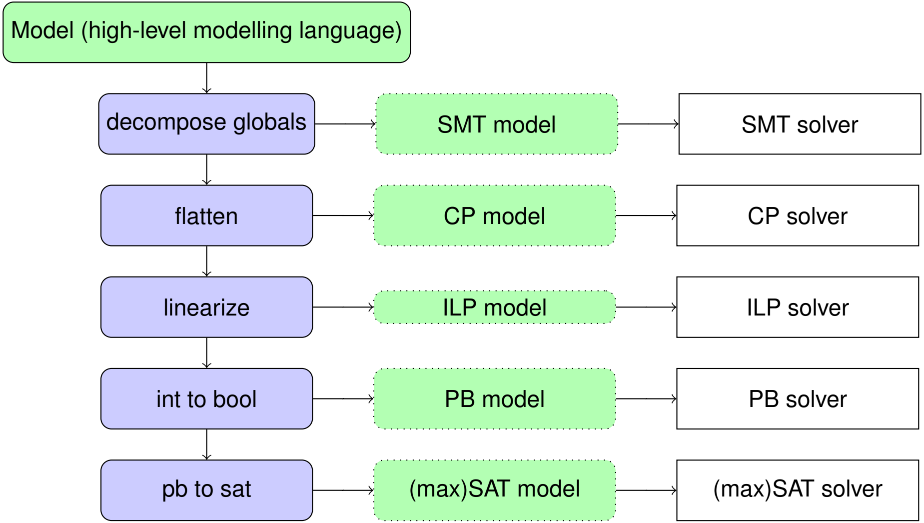
Example (Problem Definition: Sudoku)

The goal of Sudoku is to *complete* a partially filled 9x9 grid with numbers so that each row, column and 3x3 section contains each digit between 1 and 9 once.

Example (Model: Sudoku in CPMpy)

```
import cpmpy as cp
#given = np.array(...)  # load the hints, uses '0' for the empty cells
grid = cp.intvar(1,9, shape=given.shape, 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)],
    grid[given!=0] == given[given!=0],  # enforce the hints
)
model.solve()
```

From Model to Model to Solver



High-level vs low-level modelling languages

High-level modeling languages (CPMpy, MiniZinc, Essence)

- Model = list of *complex* expressions over decision variables
- Boolean logic example: (symbols are explained later)
  $(a \leftrightarrow (b \vee (c \wedge d))) \wedge (e \vee \neg f)$

Low-level modeling language (SAT, ILP, CP, ...)

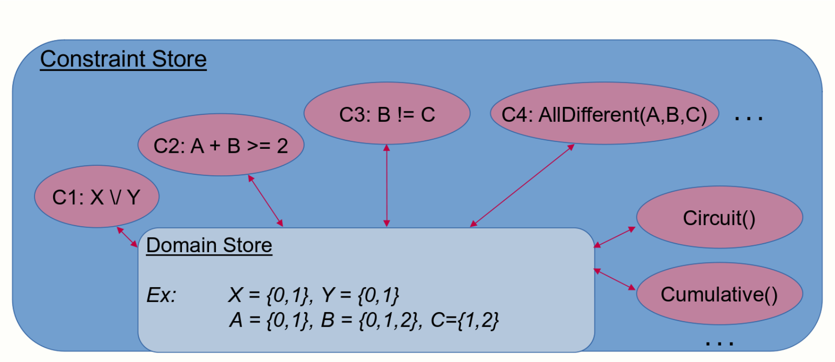
- Model = list of *atomic* constraints over decision variables
- SAT: CNF Example:
  $(a \vee \neg b) \wedge (a \vee \neg n) \wedge (\neg a \vee b \vee n) \wedge (n \vee \neg c \vee \neg d) \wedge (\neg n \vee c) \wedge (\neg n \vee d) \wedge (e \vee \neg f)$

Typically, high-level languages can translate to multiple low-level languages *(solver-agnostic)*.

This course uses 1 high-level language, all ideas translate to other languages and to low-level languages.

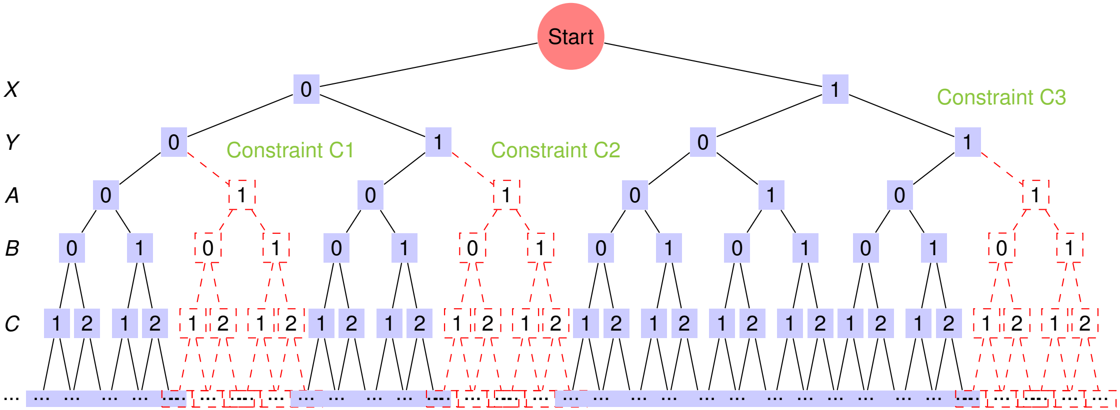
So what does solving do?

For a CP solver, a model = list of *atomic* constraints over decision variables.



It will iterate over the constraints and try to reduce domains (=propagation) and branch over variables if there is nothing left to reduce (=search)

Branching induces a search tree



Outline

Outline

Belgian Beer Tasting problem

Responsible drinking:
What beers to try, so that you can still pay attention in class tomorrow?

Tias' Belgian beer guide

Stella Artois, from Leuven,   5.2%,   must-try factor: 5/10

Duvel, devilish blond,   8.5%,   must-try factor: 8/10

Vedett IPA, tastefully hoppy,   6%,   must-try factor: 7.5/10

Tripel Karmeliet, strong blond,   8.4%,   must-try factor: 8.2/10

Gouden Carolus **Whiskey Infused**,   11.7%,   must-try factor: 9.5/10

Kriek Lindemans, sweet cherry beer,   3.5%,   must-try factor: 7/10

Belgian Beer Tasting problem

What beers to try, so that you can still pay attention in class tomorrow?

Model =

- Variables, with a domain $st, du, vi, tk, gc, kl \in \{0,1\}$

- Constraints over variables $52*st + 85*du + 60*vi + 84*tk + 117*gc + 35*kl \leq 4*52$

- Optionally: an objective maximize($50*st + 80*du + 75*vi + 82*tk + 95*gc + 70*kl$)

Model.solve()

Belgian Beer Tasting problem, CPMpy

What beers to try, so that you can still pay attention in class tomorrow?

Model =

- Variables, with a domain

- Constraints over variables

- Optionally: an objective

Model.solve()

```python
import cpmpy as cp
m = cp.Model()

st,du,vi,tk,gc,kl = cp.boolvar(shape=6)

m.add(52*st + 85*du + 60*vi + 84*tk + 117*gc + 35*kl
    <= 4*52)

m.maximize(50*st + 80*du + 75*vi + 82*tk + 95*gc +
    70*kl)


m.solve()
```

Outline

Decision variables

In this course, we only consider discrete decision variables, namely Boolean and integer decision variables:

$$
\begin{align}
b &\in \{0, 1\} \tag{1} \\
x &\in \{1, \dots, 10\} \tag{2}
\end{align}
$$

Variables have a domain, a finite set of allowed values:

- for Boolean variables, this is always $\{0, 1\}$
- for integer variables, this is specified as two parameters: *lb*=lower bound, *ub*=upper bound, domain: $\{lb..ub\}$

Sometimes, you want to create a variable with a sparse domain: $x \in \{1, 2, 5, 8, 9\}$

Some modeling languages support other variables (floats, sets, strings, bitvectors)

Outline

Logical constraints

Logical constraints involve Boolean operators over Boolean expressions

Boolean operators:

- negation: $\neg a$ (in text sometimes $-a$)
- or: $a \vee b$ (in text sometimes written as $a \mid b$)
- and: $a \wedge b$ (in text sometimes written as $a \,\&\, b$)
- equivalence: $a \leftrightarrow b$ (also called reification or double-implication)
- implication: $a \rightarrow b$ (also called half-reification)

Each operator has its own *truth table*: a/b values that make the expression true or false.

Logical constraints

Each operator has its own *truth table*: a/b values that make the expression true or false.

The one many people find tricky is the one of **implication**, called material implication in logic:

a	b	$a \to b$
T	T	T
T	F	F
F	T	T
F	F	T

Verify: $a \to b$ is equivalent to $\neg a \vee b$
Also: $a = b$ is equivalent to $(a \to b) \wedge (b \to a)$

Logical constraints

Boolean quantifiers:

- universal quantification: $\forall x \in X, (x \rightarrow y)$
- existential quantification: $\exists x \in X, (x \wedge y)$

Other Boolean operator:

- exclusive-or: $a \otimes b$ (in text sometimes a xor b)

a	b	$a \otimes b$
T	T	F
T	F	T
F	T	T
F	F	F

To think about: $a \otimes b$ is equivalent to $(a \vee b) \wedge (\neg a \vee \neg b)$, what about XOR$(a, b, c)$?

Outline

Simple comparison constraints

Simple comparisons ($=, \neq, <, \leq, >, \geq$) of an integer variable:

$$x = 3 \tag{3}$$
$$y \neq 6 \tag{4}$$
$$z < 2 \tag{5}$$

A simple comparison is a Boolean-valued expression, it can be used inside Boolean operators:

$$(x = 3) \wedge (y > 5) \tag{6}$$
$$(y \neq 6) \rightarrow (a \vee (z < 2)) \tag{7}$$

Good use of brackets () avoids ambiguity.

Outline

Arithmetic constraints

Arithmetic constraints combine arithmetic operations $(+, -, *, /)$ over integers with a comparisons $(=, \neq, <, \leq, >, \geq)$:

$$x + y = 3 \tag{8}$$
$$y - z \neq 6 \tag{9}$$
$$2 * z - (x * y + y) < 2 \tag{10}$$

Integer division $x/y == 5$ is tricky because it is undefined for $y = 0$. It is a partial function. Some languages/systems simply forbid a division where the nominator has 0 in its domain.
(there are more peculiarities with integer division, such as using *floor division* or *rounding division* for negative numbers...)

Arithmetic constraints

Linear constraints only involve arithmetic operations $(+, -)$ and a multiplication of a variable with a constant. Linear inequalities further only use the comparisons $(<, \leq, >, \geq)$:

$$x + y \geq 3 \tag{11}$$
$$y - z < 6 + x \tag{12}$$

All integer linear inequalities can be rewritten to a normal form $a_1 x_1 + \ldots + a_n x_n \leq b$

A linear equality $x + y = 5$ can be rewritten as $(x + y \leq 5) \wedge (x + y \geq 5)$

A linear dis-equality $x + y \neq 5$ leads to a disjunction: $(x + y < 5) \vee (x + y > 5)$ and cannot be rewritten to a conjunction of inequalities without adding a new variable...

Arithmetic constraints

Many other arithmetic operations exist:

- absolute value $|a|$
- modulo $x\%y$
- minimum $min(x, y, z)$
- maximum $max(x, y, z)$

These can be used in arithmetic constraints too (e.g. arithmetic operators + comparison).

Just like simple constraints, arithmetic constraints are Boolean-valued expressions too; and can be used in Boolean expressions.

A nested expression nests all sorts of Boolean operators and/or arithmetic constraints and operators. A contrived example:
$(a \vee (|x - y| > z/2)) \rightarrow (r * s \neq max(x, y - t, |z - 3|)) \wedge (b \otimes (c \leftrightarrow d))$

Outline

Global constraints

Many other constraints and operations that do not exist in standard mathematics can be defined.
In the constraint programming community, such constraints are called global constraints:

- AllDifferent(x, y, z)
- Table$([x, y], [[1, 2], [1, 4], [3, 4]])$
- Count$(X, 3) == z$
- ...

In some systems, the concept 'Count$(X, 3) == z$' is modelled with a predicate 'Count$(X, 3, z)$'.
In this course, we call the 'Count$(X, 3)$' part a global function, the integer-valued counter-part of a global constraint; such that it can be nested with arithmetic operators, e.g. $10 * (Count(X, 3) - Count(Y, 3))$

Outline

Objective functions

The objective function is an integer-valued expression that must be minimized or maximized.

Global functions are valid integer-valued expressions too, as are nested arithmetic expressions (in high-level languages at least).

Sometimes, we want to relax a hard constraint by allowing it to be violated, but penalizing that **violation in the objective**.
When we add a constraint (Boolean-valued expression) to the objective function, we call that constraint a soft constraint, e.g. $[z == 0]$ below:

$$\begin{aligned} \text{maximize} \quad & 10 * x + 3 * y + [z == 0] && (13) \\ \text{s.t.} \quad & x + y < 10, && (14) \\ & x + y + z > 5, && (15) \\ & x, y, z \in \{0..10\} && (16) \end{aligned}$$

Outline

Using CPMpy includes the following:

- Import and model creation
- Decision variables
- Constraints
- Objective function (optional)
- Solving
- Printing output

Example (Showcase)

```python
import cpmpy as cp
m = cp.Model()

# Decision variables
b = cp.boolvar(name="b")
X = cp.intvar(1,10, shape=3, name="X")

# Constraints
m.add(X[0] == 1)
m.add(cp.AllDifferent(X))
m.add(b.implies(X[1] + X[2] > 5))

# Objective function (optional)
m.maximize(cp.sum(X) + 100*b)

if m.solve():
    print(X.value(), b.value())
    print("obj:", m.objective_value())
else:
    print("No solution found.")
    print(m)
```

Install, import, create model

Single page, all you need documentation:
https://cpmpy.readthedocs.io/en/latest/modeling.html

Installing: `pip install cpmpy`

Importing and model creation:

```
import cpmpy as cp
m = cp.Model()
```

You can also `from cpmpy import *` which will override any/all/min/max/sum for convenience but this can be confusing for novices (e.g. when does `sum` compute a value or create an expression).

Decision variables

CPMpy supports discrete decision variables, namely Boolean and integer decision variables:

```
b = cp.boolvar(name="b")
x = cp.intvar(lb=1,ub=10, name="x")
```

Variables have a domain, a finite set of allowed values:

- for Boolean variables, this is implicitely $\{0, 1\}$
- for integer variables, this is specified as the first two parameters: *lb*=lower bound, *ub*=upper bound, domain: $\{lb..ub\}$

If you want a sparse domain, containing only a few values, you can:

- Add constraints to forbid specific values, e.g. `x != 3, x != 5, x != 7`
- Or use the shorthand *InDomain* global constraint:

  `cp.InDomain(x, [1,2,4,6,8,9])`

Decision variables 2/2

Decision variables have a unique name. You can set it yourself, otherwise a unique name will automatically be assigned to it. If you print decision variables, `print(b, x)`, it will print the name.

CPMpy creates *n-dimensional* NumPy arrays when creating variables!
This is very convenient for Numpy-style *vectorized* operations and for integration with machine learning libraries.

The *shape* argument allows you to specify the dimensions of the array. All variables will have the same initial domain:

```
B = cp.boolvar(shape=4, name="B")
print(B)  # [B[0] B[1] B[2] B[3]]

X = cp.intvar(1,10, shape=(2,2), name="X")
print(X)  # [[X[0,0] X[0,1]]
          #  [X[1,0] X[1,1]]]
```

Note: Numpy indexing

Python creates lists of lists, each requiring an index:

```
A = [["00","01"],
     ["10","11"]]
print(A[0][1])  # out: '01'
```

Numpy creates an array, which allows indexing with a tuple:

```
B = np.array(A)
print(B[0,1])  # out: '01'
```

Also accepts ':' which means this entire dimension:

```
print(B[0,:])  # out: ['00' '01']
```

Or using an equal sized array of Booleans (also called a 'selector'):

```
Sel = [[False, True],
       [True, False]]
print(B[Sel])  # out: ['01' '10']
```

Advanced: Vectorized operations

Because decision variables are NumPy arrays in CPMpy, you can also do vectorized operations on them: an operation on two equal sized arrays will create an (equal sized) array of element-wise operations:

Example (vectorized operations)

```
X = cp.intvar(1,9, shape=3, name="X")
A = [1,2,4]

print(X == A)  # output: [X[0]==1 X[1]==2 X[2]==4]
```

Broadcasting in NumPy allows operations between arrays of different shapes, by automatically expanding the smaller array along its dimensions to match the larger array's shape:

Example (broadcasting)

```
print(X == 3)  # output: [X[0]==3 X[1]==3 X[2]==3]
```

Expressing constraints

A constraint is an expression that is added to a model, the solver will enforce it to always be true, e.g.:

```
m.add(X[0] == 1)
m.add(cp.AllDifferent(X))
m.add(b.implies(X[1] + X[2] > 5))
```

The `m += o` is Python syntactic sugar for `m.__add__(o)`.

We will differentiate *Boolean-valued* expressions like `x[0] == 1` and *integer-valued* expressions like `x[1] + x[2]`. Only Boolean-valued expressions can be added as a constraint.

Common constraints

Example (Typical logical constraints)

```
(a,b,c) = cp.boolvar(shape=3)
m.add(a | b)   # a OR b
m.add(~(a & b))   # NOT (a AND b)
m.add(a.implies(b | c))   # a -> (b OR c)
m.add(a == b)   # equivalence: (a -> b) & (b -> a)
m.add(a != b)   # same as ~(a==b) and same as (a == ~b)
```

CPMpy overloads the Python bitwise operators &, |, ~. They have precedence over all other operators, so `a` == 0 | `b` == 1 is **wrongly** interpreted as `a` == (0 | `b`) == 1 `-- WRONG!`. So make sure to **always write explicit brackets** to express (`a` == 0) | (`b` == 1).

Example (n-ary logical constraints)

```
Bv = cp.boolvar(shape=3)
m.add(cp.any([Bv[0], Bv[1], Bv[2]]))   # explicit list
m.add(~cp.all(Bv))   # (numpy) array
```

Common constraints 2/4

Example (Typical comparison constraints)

```
b = cp.boolvar()
x = cp.intvar(0, 10)
Iv = cp.intvar(0, 10, shape=3)

m.add(x > 3)
m.add(x != 6)

m.add(Iv == 1)  # vectorized, shorthand for:
m.add([Iv[0] == 1, Iv[1] == 1, Iv[2] == 1])
```

You can not use a numeric expression as a Boolean expression, e.g. invalid: `b | x -- WRONG!`, you need to add your intended meaning of truth: `b | (x != 0)`

Common constraints 3/4

Example (Some arithmetic constraints)

```
Xs = cp.intvar(0, 10, shape=3, name="Xs")
Ys = cp.intvar(1, 10, shape=3, name="Ys")
W = np.array([1,3,-5])  # numpy array for use in vectorized multiplication

m.add(Xs[0] - Ys[0] == 5)
m.add(cp.sum(Xs) != 1)
m.add(cp.sum(W*Xs) > 3)  # 1*Xs[0] + 3*Xs[1] + (-5)*Xs[2] > 3

# arbitrary nested expressions:
m.add(3*Xs[0] < abs(5 - cp.max(Xs) + cp.min(Ys)))
```

You **can** use any Boolean expression as a numeric expression, e.g. valid:

$b + x > 2$

Common constraints 4/4

Example (Typical global constraints)

```
Ivs = cp.intvar(1,10, shape=4, name="ivs")
b = cp.boolvar()

m.add(cp.AllDifferent(Ivs))
m.add(b.implies(cp.AllEqual(Ivs)))
m.add(cp.max(Ivs) == 3)  # maximum of 'ivs' values equals 3
m.add(cp.Table(Ivs, [[1,1,2,4],  # values must match one of the rows
                     [1,2,3,6],
                     [2,3,5,10]]))
```

Many more global constraints and global functions exists! We will see them throughout the course.

Handy summary sheet: https://cpmpy.readthedocs.io/en/latest/summary.html

Objective functions (optional)

If a model has *no objective function* specified, then it is a satisfaction problem: the goal is to find out whether a solution, any solution, exists.
When an objective function is added it is an optimisation problem and this function needs to be minimized or maximized.

Example (Objective function, from showcase example)

```
# Objective function (optional)
m.maximize(cp.sum(X) + 100*b)
```

Any expression can be added as an objective function (maximize() or minimize())

CPMpy does not support multi-objective optimisation yet:
multiple objective functions must either be aggregated into a weighted sum,
or handled outside the model.

Solving and printing output

Example (Solving)

```
Xs = cp.intvar(1,10, shape=3)
m = cp.Model( cp.AllDifferent(Xs) )
m.maximize(cp.sum(Xs))

hassol = m.solve()
```

solve() accepts arguments such as `time_limit=`, `solver=` and solver-specific ones

Example (Printing output)

```
print("Status:", m.status())  # Status: ExitStatus.OPTIMAL (0.03033301 seconds)
if hassol:
    print(m.objective_value(), Xs.value())  # 27 [10  9  8]
else:
    print("No solution found.")
    print(m)  # pretty-prints the constraints in the model
```

Focus point: reification

Reification enables the reasoning about the truth of a constraint or a Boolean expression.

Example

constraint `x < y`
requires that `x` be smaller than `y`.
constraint `b == (x < y)` requires that the Boolean variable `b` takes the value `True` iff `x` is smaller than `y`:
the constraint `x < y` is said to be *reified*, and `b` is called its *reified variable*.

Reification is a powerful mechanism that enables:

- efficient reuse of logical components through their reified variable;
- higher-level modelling (e.g. nested expressions, soft constraints)

Example (Soft Constraints: Alignment Photo Problem)

A set of students want to line up for a class photo.

Consider:

`Wishes` = [(`"Dimos"`, `"Stella"`), (`"Marco"`, `"Dimos"`), ...]
where each pair *(who,whom)* denotes that student *who* wants to be next to student *whom* on the photo.
Maximise the number of granted wishes.

Let decision variable `Pos[s]` denote the position in 0..`len(Students)` of student `s` on the photo.

The array `Pos` must form a permutation of the positions:

```
m = cp.Model( cp.AllDifferent(Pos) )
```

The objective, formulated using nested expressions, is:

```
m.maximize(cp.sum([ cp.abs(Pos[who] - Pos[whom]) == 1
                    for (who,whom) in Wishes ]))
```

Constraint `cp.abs(Pos[who] - Pos[whom]) == 1` will automatically be reified.

Example (Soft Constraints: Weighted Alignment Photo Problem)

A set of students want to line up for a class photo.

Consider:

`Wishes` = [(`"Dimos"`, `"Stella"`, 2), (`"Marco"`, `"Dimos"`, 1), ...]
where each pair *(who,whom,bid)* denotes that student *who* wants to bid *bid* to be next to student *whom* on the photo.
Maximise the weighted number of granted wishes.

Let decision variable `Pos[s]` denote the position in 0..`len(Students)` of student `s` on the photo.

The array `Pos` must form a permutation of the positions:

```
m = cp.Model( cp.AllDifferent(Pos) )
```

The objective, formulated using nested expressions, is:

```
m.maximize(cp.sum([bid*(cp.abs(Pos[who] - Pos[whom]) == 1)
                   for (who,whom,bid) in Wishes]))
```

General-purpose Modelling Languages

- CPMpy: `https://cpmpy.readthedocs.io/`
- MiniZinc: `https://www.minizinc.org`
- Essence and Essence′: `https://constraintmodelling.org`
- OPL: `https://www.ibm.com/optimization-modeling`
- SMT-lib: `https://smtlib.cs.uiowa.edu`
- AIMMS: `https://aimms.com`
- AMPL: `https://ampl.com`
- GAMS: `https://gams.com`
- FICO Xpress Insight: `https://www.fico.com/en/products/fico-xpress-optimization`
- Comet: `https://mitpress.mit.edu/books/constraint-based-local-search`
- ...