CPMpy,
a numpy-based CP modeling environment

Prof. Tias Guns
<tias.guns@kuleuven.be>
@TiasGuns

CP 2021 tutorial

https://cpmpy.readthedocs.io/

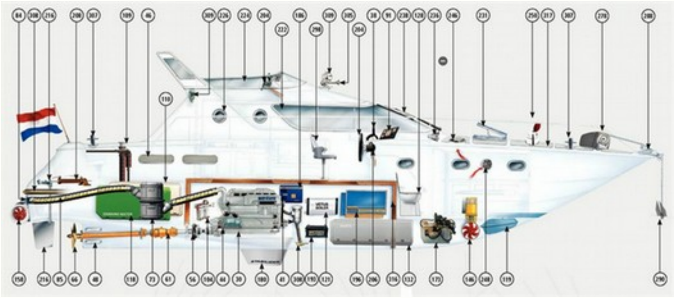
CP: Constraint Programming

"Solving *constrained* optimisation problems"

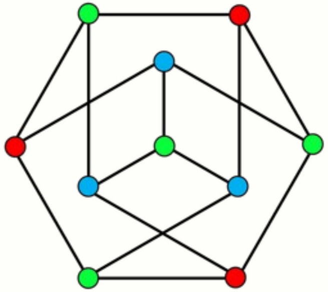
- Vehicle Routing

- Scheduling

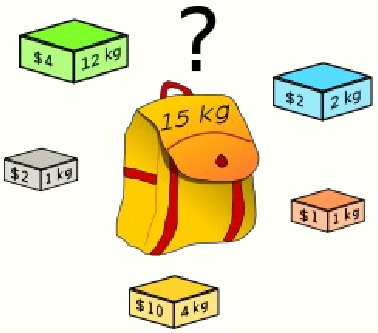
- Configuration



- Graph problems & mathematical puzzles



CP Modeling



Knapsack:

Model =

- Variables, with a domain
- Constraints over variables
- Optionally: an objective

- gr, bl, og, ye, gy :: {0,1}
- 12*gr + 2*bl + 1*og + 4*ye + 1*gy <= 15
- maximize(4*gr + 2*bl + 1*og + 10*ye + 2*gy)

Model.solve()

CPMpy: Constraint Programming and Modeling in Python

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

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

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

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

Getting started:

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

User Documentation:

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

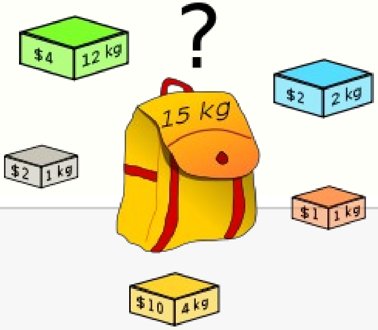
API documentation:

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

https://cpmpy.readthedocs.io/

CP Modeling

Knapsack:



Model =

- Variables, with a domain
- Constraints over variables
- Optionally: an objective

Model.solve()

```
model = Model()

gr,bl,og,ye,gy = boolvar(shape=5)

model += (12*gr + 2*bl + 1*og + 4*ye + 1*gy <= 15)

model.maximize(4*gr + 2*bl + 1*og + 10*ye + 2*gy)


model.solve()
```

```
print(gr.value(), bl.value(), og.value(), ye.value(), gy.value())
```

```
0 1 1 1 1
```

Why CPMpy?

Holy Grail: user specifies, solver solves [Freuder,1997]

I think we reached it… MiniZinc, Essence’

Why CPMpy?

Holy Grail: user specifies, solver solves [Freuder,1997]

I think we reached it… MiniZinc, Essence’

“Beyond NP” → CP as an oracle

Existing trend in AI, often with SAT or SMT or MIP solvers.

Growing need for more high-level solvers :: CP

Why CPMpy?

The more practical side of the story:

```
model = Model()

gr,bl,og,ye,gy = boolvar(shape=5)

model += (12*gr + 2*bl + 1*og + 4*ye + 1*gy <= 15)

model.maximize(4*gr + 2*bl + 1*og + 10*ye + 2*gy)


model.solve()
```

You have your CP model, but…

- Where does the data come from? (python code)
- How can you graphically visualize the result? (python code)
- How do you compare different formulations (python code)
- How can you test what solver params work best (python code)
- How do you make it solve for predicted cost vectors (python code)

What is CPMpy?

CPMpy is a:

- CP modeling library in Python
- based on numpy
- with direct solver access

CPMpy quick start

```
> pip3 install cpmpy
```

Will also install ortools (CP-SAT is default solver)

```
> python3
from cpmpy import *
a = boolvar(); b = boolvar()
Model( a & ~b ).solve()
```

returns ‘True’

Python/numpy how?

- All variables are numpy tensors
- Operator overloading
- Array indexing

```
v = boolvar()
print(v)
```

```
BV5
```

CPMpy variables are numpy arrays

CPMpy variables are numpy arrays

```python
v = boolvar()
print(v)
```

```
BV5
```

```python
v = boolvar(shape=5)
print(v)
```

```
[BV6 BV7 BV8 BV9 BV10]
```

CPMpy variables are numpy arrays

```
v = boolvar()
print(v)
```

```
BV5
```

```
v = boolvar(shape=5)
print(v)
```

```
[BV6 BV7 BV8 BV9 BV10]
```

```
v = intvar(1,9, shape=5)
print(v)
```

```
[IV7 IV8 IV9 IV10 IV11]
```

CPMpy variables are numpy arrays

```
v = boolvar()
print(v)
```

```
BV5
```

```
v = boolvar(shape=5)
print(v)
```

```
[BV6 BV7 BV8 BV9 BV10]
```

```
v = intvar(1,9, shape=5)
print(v)
```

```
[IV7 IV8 IV9 IV10 IV11]
```

```
m = intvar(1,9, shape=(3,3))
print(m)
```

```
[[IV12 IV13 IV14]
 [IV15 IV16 IV17]
 [IV18 IV19 IV20]]
```

CPMpy variables are numpy arrays

```
v = boolvar()
print(v)
```

```
BV5
```

```
v = boolvar(shape=5)
print(v)
```

```
[BV6 BV7 BV8 BV9 BV10]
```

```
v = intvar(1,9, shape=5)
print(v)
```

```
[IV7 IV8 IV9 IV10 IV11]
```

```
m = intvar(1,9, shape=(3,3))
print(m)
```

```
[[IV12 IV13 IV14]
 [IV15 IV16 IV17]
 [IV18 IV19 IV20]]
```

```
t = boolvar(shape=(2,3,4))
print(t)
```

```
[[[BV11 BV12 BV13 BV14]
  [BV15 BV16 BV17 BV18]
  [BV19 BV20 BV21 BV22]]

 [[BV23 BV24 BV25 BV26]
  [BV27 BV28 BV29 BV30]
  [BV31 BV32 BV33 BV34]]]
```

CPMpy variables are numpy arrays

```
v = boolvar()
print(v)
```

```
BV5
```

```
v = boolvar(shape=5)
print(v)
```

```
[BV6 BV7 BV8 BV9 BV10]
```

```
v = intvar(1,9, shape=5)
print(v)
```

```
[IV7 IV8 IV9 IV10 IV11]
```

```
m = intvar(1,9, shape=(3,3))
print(m)
```

```
[[IV12 IV13 IV14]
 [IV15 IV16 IV17]
 [IV18 IV19 IV20]]
```

```
t = boolvar(shape=(2,3,4))
print(t)
```

```
[[[BV11 BV12 BV13 BV14]
  [BV15 BV16 BV17 BV18]
  [BV19 BV20 BV21 BV22]]

 [[BV23 BV24 BV25 BV26]
  [BV27 BV28 BV29 BV30]
  [BV31 BV32 BV33 BV34]]]
```

```
puzzle_start = np.array([
    [0,3,6],
    [2,4,8],
    [1,7,5]])

(dim,dim2) = puzzle_start.shape
assert (dim == dim2), "puzzle needs square shape"
n = dim*dim2 - 1 # e.g. an 8-puzzle

# State of puzzle at every step
K = 20
x = intvar(0,n, shape=(K,dim,dim), name="x")
print(x)
```

```
[[[x[0,0,0] x[0,0,1] x[0,0,2]]
  [x[0,1,0] x[0,1,1] x[0,1,2]]
  [x[0,2,0] x[0,2,1] x[0,2,2]]]

 [[x[1,0,0] x[1,0,1] x[1,0,2]]
```

Operator overloading

```
x,y,z = intvar(1,9, shape=3)
print( x + y )
```

```
(IV33) + (IV34)
```

```
print( x * y )
```

```
(IV33) * (IV34)
```

```
print( abs(x - y) )
```

```
abs([(IV33) + (-(IV34))])
```

```
a = intvar(1,9, shape=5, name="a")
print( sum(a) )
```

```
sum([a[0], a[1], a[2], a[3], a[4]])
```

```
c = ( abs(sum(a) - (x+y)) == z )
print( c )
```

```
(abs([sum([a[0], a[1], a[2], a[3], a[4], -((IV33) + (IV34))])])) == (IV35)
```

Operator overloading

```
x,y,z = intvar(1,9, shape=3)
print( x + y )
```

```
(IV33) + (IV34)
```

```
print( x * y )
```

```
(IV33) * (IV34)
```

```
print( abs(x - y) )
```

```
abs([(IV33) + (-(IV34))])
```

```
a = intvar(1,9, shape=5, name="a")
print( sum(a) )
```

```
sum([a[0], a[1], a[2], a[3], a[4]])
```

```
c = ( abs(sum(a) - (x+y)) == z )
print( c )
```

```
(abs([sum([a[0], a[1], a[2], a[3], a[4], -((IV33) + (IV34))])])) == (IV35)
```

```
type(c)
```

```
cpmpy.expressions.core.Comparison
```

```
c.name
```

```
'=='
```

```
c.args[1]
```

```
IV35
```

```
type(c.args[0])
```

```
cpmpy.expressions.core.Operator
```

```
c.args[0].name
```

```
'abs'
```

Array indexing

```
x = boolvar(shape=(4,4), name="x")
print(x)
```

```
[[x[0,0] x[0,1] x[0,2] x[0,3]]
 [x[1,0] x[1,1] x[1,2] x[1,3]]
 [x[2,0] x[2,1] x[2,2] x[2,3]]
 [x[3,0] x[3,1] x[3,2] x[3,3]]]
```

```
print(x[0,:])
```

```
[x[0,0] x[0,1] x[0,2] x[0,3]]
```

```
print(x[:,0])
```

```
[x[0,0] x[1,0] x[2,0] x[3,0]]
```

```
print(x[:,1:-1])
```

```
[[x[0,1] x[0,2]]
 [x[1,1] x[1,2]]
 [x[2,1] x[2,2]]
 [x[3,1] x[3,2]]]
```

Python’s indexing

Array indexing

```
x = boolvar(shape=4, name="x")
print(x)
```

```
[x[0] x[1] x[2] x[3]]
```

```
sel = np.array([True, False, True, False])
print(x[sel])
```

```
[x[0] x[2]]
```

```
print(x[np.arange(4) % 2 == 0])
```

```
[x[0] x[2]]
```

Numpy’s indexing

Vectorized operations

```
x = intvar(1,9, shape=3, name="x")
y = intvar(1,9, shape=3, name="y")
print(x + y)
```

```
[(x[0]) + (y[0]) (x[1]) + (y[1]) (x[2]) + (y[2])]
```

```
print(x == [1,2,3])
```

```
[x[0] == 1 x[1] == 2 x[2] == 3]
```

Numpy's operator overloading

```
print(x == 1)
```

```
[x[0] == 1 x[1] == 1 x[2] == 1]
```

Numpy's broadcasting

Python/numpy how?

- Every variable is a numpy tensor
- Operator overloading
- Array indexing

Let’s see it in action...

Sudoku

```
import numpy as np
from cpmpy import *

e = 0 # value for empty cells
given = np.array([
    [e, e, e,  2, e, 5,  e, e, e],
    [e, 9, e,  e, e, e,  7, 3, e],
    [e, e, 2,  e, e, 9,  e, 6, e],

    [2, e, e,  e, e, e,  4, e, 9],
    [e, e, e,  e, 7, e,  e, e, e],
    [6, e, 9,  e, e, e,  e, e, 1],

    [e, 8, e,  4, e, e,  1, e, e],
    [e, 6, 3,  e, e, e,  e, 8, e],
    [e, e, e,  6, e, 8,  e, e, e]])


# Variables
puzzle = intvar(1,9, shape=given.shape, name="puzzle")
```

Classic sudoku

```
model = Model()
n = given.shape[0]

# Constraints on rows and columns
for i in range(n):
    model += AllDifferent([puzzle[i,j] for j in range(n)])
    model += AllDifferent([puzzle[j,i] for j in range(n)])

# Constraints on blocks
for i in range(0,9, 3):
    for j in range(0,9, 3):
        model += AllDifferent([puzzle[r,c]
                               for r in range(i,i+3)
                               for c in range(j,j+3)])

# Constraints on values (cells that are not empty)
for r in range(n):
    for c in range(n):
        if given[r,c] != e:
            model += puzzle[r,c] == given[r,c]

model.solve()
```

CPMpy sudoku

```
model = Model()


# Constraints on rows and columns
model += [AllDifferent(row) for row in puzzle]
model += [AllDifferent(col) for col in puzzle.T]


# Constraints on blocks
for i in range(0,9, 3):
    for j in range(0,9, 3):
        model += AllDifferent(puzzle[i:i+3, j:j+3])


# Constraints on values (cells that are not empty)
model += [puzzle[given!=e] == given[given!=e]]


model.solve()
```

Job shop scheduling

```
jobs_data = cpm_array([ # (job, machine) = duration
    [3,2,2], # job 0
    [2,1,4], # job 1
    [0,4,3], # job 2 (duration 0 = not used)
])
max_dur = sum(jobs_data.flat)

n_jobs, n_machines = jobs_data.shape
all_jobs = range(n_jobs)
all_machines = range(n_machines)


# Variables
start_time = intvar(0, max_dur, shape=(n_machines,n_jobs), name="start")
end_time = intvar(0, max_dur, shape=(n_machines,n_jobs), name="stop")
```

Classic jobshop

```
model = Model()
# end = start + dur
for j in all_jobs:
    for m in all_machines:
        model += (end_time[m,j] == start_time[m,j] + jobs_data[j,m])

# Precedence constraint per job
for j in all_jobs:
    for m1,m2 in combinations(all_machines,2): # [0,1,2]->[(0,1),(0,2),(1,2)]
        model += (end_time[m1,j] <= start_time[m2,j])

# No overlap constraint: one starts before other one ends
for m in all_machines:
    for j1,j2 in combinations(all_jobs, 2):
        model += (start_time[m,j1] >= end_time[m,j2]) | \
                 (start_time[m,j2] >= end_time[m,j1])

# Objective: makespan
makespan = Maximum([end_time[m,j] for m in all_machines for j in all_jobs])
model.minimize(makespan)
model.solve()
```

Classic jobshop

```
model = Model()
# end = start + dur
for j in all_jobs:
    for m in all_machines:
        model += (end_time[m,j] == start_

# Precedence constraint per job
for j in all_jobs:
    for m1,m2 in combinations(all_machine
        model += (end_time[m1,j] <= start

# No overlap constraint: one starts befor
for m in all_machines:
    for j1,j2 in combinations(all_jobs, 2
        model += (start_time[m,j1] >= end
                  (start_time[m,j2] >= end

# Objective: makespan
makespan = Maximum([end_time[m,j] for m i
model.minimize(makespan)
model.solve()
```

CPMpy jobshop

```
model = Model()
# end = start + dur
model += (end_time == start_time + jobs_data.T)



# Precedence constraint per job
for m1,m2 in combinations(all_machines,2):
    model += (end_time[m1,:] <= start_time[m2,:])


# No overlap constraint: one starts before other one ends
for j1,j2 in combinations(all_jobs, 2):
    model += (start_time[:,j1] >= end_time[:,j2]) | \
             (start_time[:,j2] >= end_time[:,j1])


# Objective: makespan
makespan = max(end_time)
model.minimize(makespan)
model.solve()
```

Let’s inspect the result...

```
print("Makespan:",makespan.value())
print("Schedule:")
grid = -8*np.ones((n_machines, makespan.value()), dtype=int)

for j in all_jobs:
    for m in all_machines:
        grid[m,start_time[m,j].value():end_time[m,j].value()] = j
print(grid)
```

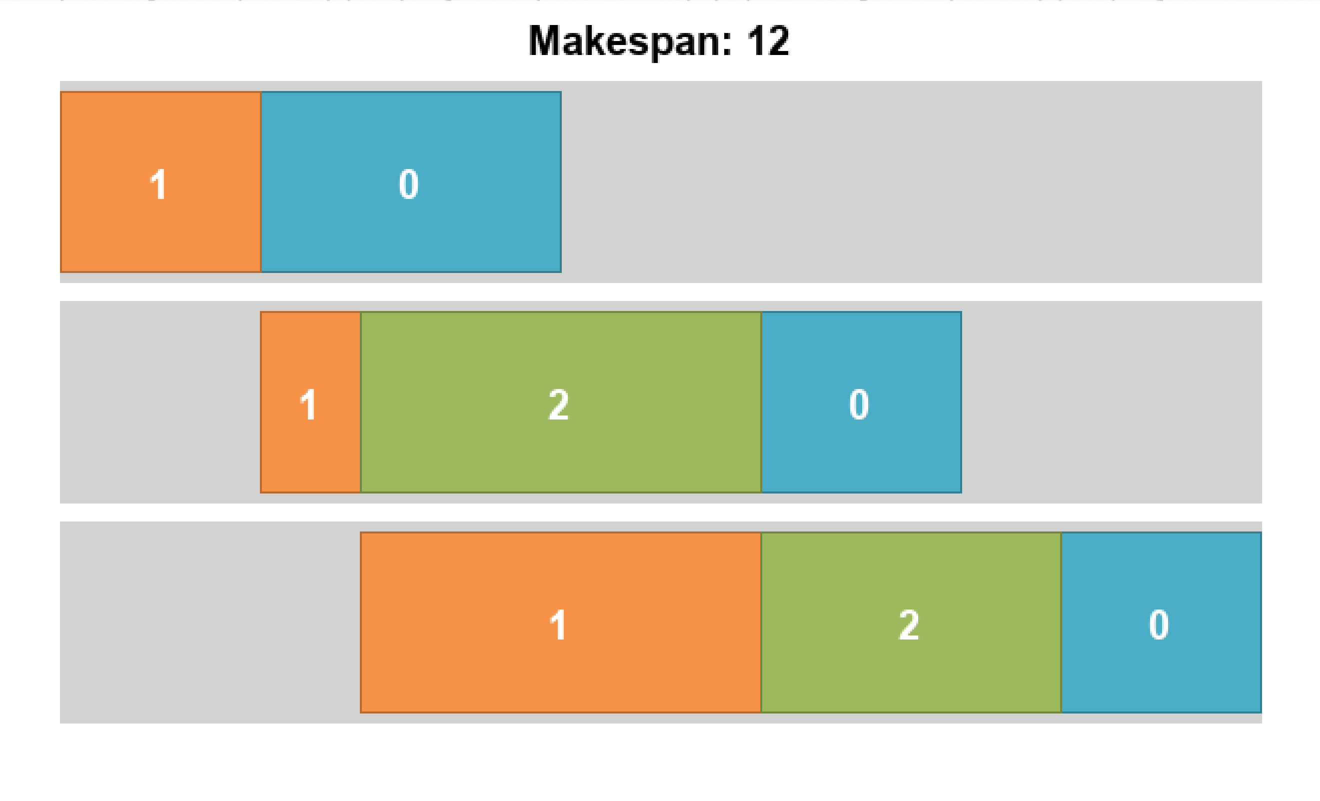
```
Makespan: 12
Schedule:
[[ 1  1  0  0  0 -8 -8 -8 -8 -8 -8 -8]
 [-8 -8  1  2  2  2  2  0  0 -8 -8 -8]
 [-8 -8 -8  1  1  1  1  2  2  2  0  0]]
```

Let’s **visualize** the result...

```
from PIL import Image, ImageDraw, ImageFont

# based on Alexander Schiendorfer's https://github.com/Alexander-Schiendorfer/cp-examples
def visualize_scheduling(start, end):
    nMachines, nJobs = start.shape
    makespan = max(end.value())

    # Draw solution
    # Define start location image & unit sizes
    start_x, start_y = 30, 40
    pixel_unit = 50
    pixel_task_height = 100
    vert_pad = 10

    imwidth, imheight = makespan * pixel_unit + 2 * start_x, start_y + start_x + nMachine

    # Create new Image object
    img = Image.new("RGB", (imwidth, imheight), (255, 255,255))

    # Create rectangle image
    img1 = ImageDraw.Draw(img)

    # Get a font
    try:
        myFont = ImageFont.truetype("arialbd.ttf", 20)
    except:
        myFont = ImageFont.load_default()

    # Draw makespan label
```

Let’s **visualize** the result...

```
from PIL import Image, ImageDraw, ImageFont
```



N-puzzle (planning as SAT)

```python
# '0' is empty spot
puzzle_start = np.array([
    [3,7,5],
    [1,6,4],
    [8,2,0]]) # 19 steps
puzzle_end = np.array([
    [1,2,3],
    [4,5,6],
    [7,8,0]])

def n_puzzle(puzzle_start, puzzle_end, K):
    print("Max steps:", K)
    m = Model()

    (dim,dim2) = puzzle_start.shape
    assert (dim == dim2), "puzzle needs square shape"
    n = dim*dim2 - 1 # e.g. an 8-puzzle

    # State of puzzle at every step
    x = intvar(0,n, shape=(K,dim,dim), name="x")
```

```python
    # Start state constraint
    m += (x[0] == puzzle_start)

    # End state constraint
    m += (x[-1] == puzzle_end)

    # define neighbors = allowed moves for the '0'
    def neigh(i,j):
        # same, left,right, down,up, if within bounds
        for (rr, cc) in [(0,0),(-1,0),(1,0),(0,-1),(0,1)]:
            if 0 <= i+rr and i+rr < dim and 0 <= j+cc and j+cc < dim:
                yield (i+rr,j+cc)

    # Transition: define next (t) based on prev (t-1) + invariants
    for t in range(1, K):
        # Invariant: in each step, all cells are different
        m += AllDifferent(x[t])

        # Invariant: only the '0' position can move
        m += ((x[t-1] == x[t]) | (x[t-1] == 0) | (x[t] == 0))

        # for each position, determine reachability of the '0' position
        for i in range(dim):
            for j in range(dim):
                m += (x[t,i,j] == 0).implies(any(x[t-1,r,c] == 0 for r,c in neigh(i,j)))

    return (m,x)
```

N-puzzle (planning as SAT)

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 10)
m.solve()
print(m.status())
```

```
Max steps: 10
ExitStatus.UNSATISFIABLE (0.010442393000000001 seconds)
```

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 100)
m.solve()
print(m.status())
```

```
Max steps: 100
ExitStatus.OPTIMAL (2.20881433 seconds)
```

N-puzzle (planning as SAT)

```
K0 = 5
step = 4

(m,x) = n_puzzle(puzzle_start, puzzle_end, K0)
while not m.solve():
    print(m.status())
    K0 = K0 + step
    (m,x) = n_puzzle(puzzle_start, puzzle_end, K0)

print(m.status())
```

```
Max steps: 5
ExitStatus.UNSATISFIABLE (0.0022607060000000003 seconds)
Max steps: 9
ExitStatus.UNSATISFIABLE (0.008280570000000001 seconds)
Max steps: 13
ExitStatus.UNSATISFIABLE (0.012872023000000002 seconds)
Max steps: 17
ExitStatus.UNSATISFIABLE (0.036358970000000004 seconds)
Max steps: 21
ExitStatus.OPTIMAL (0.113619127 seconds)
```

Other ways to make it faster?

https://github.com/CPMpy/cpmpy/blob/master/examples/advanced/hyperparameter_search.py

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 20)

from cpmpy.solvers import CPM_ortools, param_combinations

all_params = {'cp_model_probing_level': [0,1,2,3],
              'linearization_level': [0,1],
              'symmetry_level': [0,1,2]}

configs = [] # (runtime, param)
for params in param_combinations(all_params):
    s = CPM_ortools(m)
    print("Running", params, end='\r')
    s.solve(**params)
    configs.append( (s.status().runtime, params) )

best = sorted(configs)[0]
print("\nFastest in", round(best[0],4), "seconds, config:", best[1])
```

```
Max steps: 20
Running {'cp_model_probing_level': 3, 'linearization_level': 1, 'symmetry_level': 2}
Fastest in 0.078 seconds, config: {'cp_model_probing_level': 0, 'linearization_level': 1, 'symmetry_level': 0}
```

Other ways to make it faster?

https://github.com/CPMpy/cpmpy/blob/master/examples/advanced/hyperparameter_search.py

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 20)

from cpmpy.solvers import CPM_ortools, param_combinations

all_params = {'cp_model_probing_level': [0,1,2,3],
              'linearization_level': [0,1],
              'symmetry_level': [0,1,2]}

configs = [] # (runtime, param)
```

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 100)
m.solve()
print(m.status())
```

```
Max steps: 100
ExitStatus.OPTIMAL (2.20881433 seconds)
```

```
(m,x) = n_puzzle(puzzle_start, puzzle_end, 100)
m = CPM_ortools(m)
m.solve(**best[1])
print(m.status())
```

```
Max steps: 100
ExitStatus.OPTIMAL (0.41233498100000004 seconds)
```

```
Max steps: 20
Running {'cp_model_probing_level': 3, 'linearization_level': 1, 'symmetry_level': 2}
Fastest in 0.078 seconds, config: {'cp_model_probing_level': 0, 'linearization_level': 1, 'symmetry_level': 0}
```

CPMpy tutorial

End part 1:

CPMpy is a:

CP modeling library in Python

based on numpy

with direct solver access

CPMpy,
a numpy-based CP modeling environment

Prof. Tias Guns
<tias.guns@kuleuven.be>
@TiasGuns

CP 2021 tutorial

https://cpmpy.readthedocs.io/

CPMpy tutorial

Part 1:

CPMpy is a:

CP modeling library in Python

based on numpy

with direct solver access

Part 2: CP as an oracle, repeated solving

Why CPMpy?

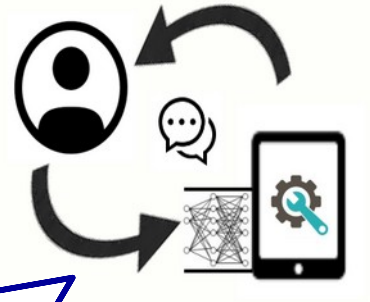
“CP as an oracle”

European Research Council
Established by the European Commission

Consolidator grant (2021-2025)

“Conversational Human-aware Technology for Optimisation”

CHAT-Opt:
Conversational **H**uman-**A**ware **T**echnology for **Opt**imisation



Towards **co-creation** of constrained optimisation solutions

- Solver that learns from user and environment
- Towards conversational: explanations and stateful interaction

https://people.cs.kuleuven.be/~tias.guns
@TiasGuns

Hiring post-docs!

Why CPMpy?

"CP as an oracle"

1. Explanation methods
2. Integrated solving and learning

=> requires repeated solving

Multiple solutions

MiniSearch-style:

```
x = intvar(0,3, shape=2)
m = Model(x[0] > x[1])

while m.solve():
    print(x.value())
    m += ~all(x == x.value()) # block solution
```

```
[3 0]
[3 1]
[3 2]
[2 0]
[1 0]
[2 1]
```

Returns True (sol. found) or False (no solution)

Adds constraint to model (even if already solved before)

Diverse solutions

```python
# a diversity measure, hamming distance
def hamm(x, y):
    return sum(x != y)

x = intvar(0,3, shape=2)
m = Model(x[0] > x[1])

store = []
while m.solve():
    print(len(store), ":", x.value())
    m += ~all(x == x.value()) # block solution
    store.append(x.value())
    # maximize number of elements that are different
    m.maximize(sum(hamm(x, sol) for sol in store))
```

```
0 : [3 0]
1 : [2 1]
2 : [1 0]
3 : [3 2]
4 : [2 0]
5 : [3 1]
```

Can change obj. function (even if already solved before)

Lazy vs Eager

```
x = intvar(0,30, shape=30)
m = Model([x[i-1] < x[i] for i in range(1, len(x))])

t0 = time.time()

while m.solve():
    print(".",end="")
    m += ~all(x == x.value()) # block solution
print("time:", time.time()-t0)
```

```
..............................time: 1.4444539546966
```

```
x = intvar(0,30, shape=30)
m = Model([x[i-1] < x[i] for i in range(1, len(x))])

t0 = time.time()
m = CPM_ortools(m)
while m.solve():
    print(".",end="")
    m += ~all(x == x.value()) # block solution
print("time:", time.time()-t0)
```

```
..............................time: 0.61836171150207
```

Model() = Lazy

Stores CPMpy expressions

SolverInterface() = Eager

Posts constraints to solver

1. Explanation methods

- Multiple and diverse solutions
- **Minimal Unsatisfiable Subset (MUS)**
- MUS/MSS enumeration (Marco algorithm)
- Optimal Unsatisfiable Subset (OUS)
- Explaining SAT problems step-wise
- Counter-factually explaining OPT problems

```python
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]))
]


i = 0 # we wil dynamically shrink mus_vars
while i < len(mus_cons):
    # add all other remaining constraints
    assum_cons = mus_cons[:i] + mus_cons[i+1:]

    if Model(assum_cons).solve():
        # 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 = assum_cons
```

```
        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])))
```

```python
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
]
```

```python
i = 0 # we wil dynamically shrink mus_vars
while i < len(mus_cons):
    # add all other remaining constraints
    assum_cons = mus_cons[:i] + mus_cons[i+1:]

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

```
        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])) -> (
```

```python
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 wil 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])
```

MUS/MSS enumeration (Marco algorithm)

```python
from marco_musmss_enumeration import SubsetSolver, MapSolver

def do_marco(model):
    sub_solver = SubsetSolver(model.constraints)
    map_solver = MapSolver(len(model.constraints))

    while True:
        seed = map_solver.next_seed()
        if seed is None:
            # all MUS/MSS enumerated
            return

        if sub_solver.check_subset(seed):
            MSS = sub_solver.grow(seed)
            yield ("MSS", [model.constraints[i] for i in MSS])
            map_solver.block_down(MSS)
        else:
            seed = sub_solver.seed_from_core()
            MUS = sub_solver.shrink(seed)
            yield ("MUS", [model.constraints[i] for i in MUS])
            map_solver.block_up(MUS)
```

MUS/MSS enumeration (Marco algorithm)

```
for kind, exprs in do_marco(m):
    print(kind,":")
    for e in sorted(exprs):
        print("\t", e)
```

```
MUS :
         (x[2]) > (x[0])
         (x[1]) > (x[2])
         (x[0]) > (x[1])
MSS :
         (x[1]) > (x[2])
         (x[0]) > (x[1])
         (((x[3]) > (x[1])) -> ((x[3]) > (x[2]))) and ((x[3] == 3) or ((x[1]) == (x[2])))
         (x[3]) > (x[0])
MSS :
         (x[0]) > (x[1])
         (((x[3]) > (x[1])) -> ((x[3]) > (x[2]))) and ((x[3] == 3) or ((x[1]) == (x[2])))
         (x[3]) > (x[0])
         (x[2]) > (x[0])
MSS :
         (((x[3]) > (x[1])) -> ((x[3]) > (x[2]))) and ((x[3] == 3) or ((x[1]) == (x[2])))
         (x[3]) > (x[0])
         (x[2]) > (x[0])
         (x[1]) > (x[2])
```

1. Explanation methods

- Multiple and diverse solutions
- Minimal Unsatisfiable Subset (MUS)
- MUS/MSS enumeration (Marco algorithm)
- **Optimal Unsatisfiable Subset (OUS)**
- Explaining SAT problems step-wise
- Counter-factually explaining OPT problems

Opt. US: implicit hitting set algorithm

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


hitset_solver = CPM_ortools(Model(
                    minimize=sum(weights*ind)))

while(True):
    hitset_solver.solve()

    # Get hitting set
    hs = ind[ind.value() == 1]

    if not assum_solver.solve(assumptions=hs):
        print("Found Optimal US, total weight:", sum(weights[ind.value() == 1]))
        for i in (ind.value() == 1).nonzero()[0]:
            print("\t", mus_cons[i], "w=",weights[i])
        break

    # hs is satisfiable subset, hit one from complement
    C = ind[ind.value() == 0]
    hitset_solver += (sum(C) >= 1)
```

Explaining SAT problems step-wise

```
b = boolvar(3, name="b")
m = Model(
    b[1].implies(b[0] | b[2]),
    b[0] | b[1],
    ~b[0],
)
m.solve()

from ocus_explanations import explain_ocus
r = explain_ocus(m.constraints, verbose=True)
```

```
Solution intersection: {b[1], ~b[0], b[2]}

Constraint(s): [~b[0]]
  and fact(s): []
          ==> ~b[0]    (cost: 6)

Constraint(s): [(b[0]) or (b[1])]
  and fact(s): [~b[0]]
          ==> b[1]     (cost: 7)

Constraint(s): [(b[1]) -> ((b[0]) or (b[2]))]
  and fact(s): [~b[0], b[1]]
          ==> b[2]     (cost: 8)
```

1. Explanation methods

- Multiple and diverse solutions
- Minimal Unsatisfiable Subset (MUS)
- MUS/MSS enumeration (Marco algorithm)
- Optimal Unsatisfiable Subset (OUS)
- Explaining SAT problems step-wise
- **Counter-factually explaining OPT problems**

```
[tias@isha]advanced python3 counterfactual_explain.py
Solution to the following knapsack problem
Values = [45 48 65 68 68 10 84 22 37 88]
Weights =  [71 89 89 13 59 66 40 88 47 89]
Capacity: 325

is: [0 3 4 6 8 9]
Resulting in an objective value of 390
Capacity used: 319


========== User Query ==========
I would like to change the following to the knapsack you provided:
Leave out items 3,4,6
Put in items 2,7
How should the values corresponding to these items change to make this assignment optimal?


========== Solving the master problem ==========
Iteration 1, candidate costs: [45 48 65 68 68 10 84 22 37 88]
  Is foil-based solution now optimal? 212 >=? 390
Iteration 2, candidate costs: [45 48 91 68  0 10  0 22 37 88]
  Is foil-based solution now optimal? 238 >=? 329
Iteration 3, candidate costs: [ 45  48  65  68  68  10  84 200  37  88]
  Is foil-based solution now optimal? 390 >=? 508
Iteration 4, candidate costs: [45 48 65  0 68 10 34 82 37 88]
  Is foil-based solution now optimal? 272 >=? 309
Iteration 5, candidate costs: [45 48 65  0 65 10  0 45 37 88]
  Is foil-based solution now optimal? 235 >=? 263
Iteration 6, candidate costs: [ 45  48 152  31  68  10  84  76  37  88]
  Is foil-based solution now optimal? 353 >=? 431
Iteration 7, candidate costs: [45 48 74  0 37 10 37 45 37 88]
  Is foil-based solution now optimal? 244 >=? 273
Iteration 8, candidate costs: [45 48 74 29 66 10  8 74 37 88]
  Is foil-based solution now optimal? 273 >=? 302
Iteration 9, candidate costs: [45 48 65 15 37 10 22 60 37 88]
  Is foil-based solution now optimal? 250 >=? 250

Values [45 48 65 15 37 10 22 60 37 88] results in an optimal solution satisfying the user query
Optimal knapsack satisfying user query: [2 7 8 9], value 250
  diff: [  0   0   0  53  31   0  62 -38   0   0]
```

Reimplementation of

A. Korikov & C. Beck, Counterfactual Explanations via Inverse Constraint Programming, CP2021

```python
# FROM examples/advanced/counterfactual_explain.py
# cutting plane algorithm

def inverse_optimize(d_orig, weights, capacity, x_d, foil_idx):
    """
    Master problem: iteratively find better values for the 'd_orig' vector
    (Korikov, A., & Beck, J. C., Counterfactual Explanations via Inverse Constraint Programming (CP2021))
    """
    master_model, d, x = make_master_problem(d_orig, weights, capacity, x_d, foil_idx)
    sub_model, x_0 = make_sub_problem(d_orig, weights, capacity)

    i = 1
    while master_model.solve() is not False:
        d_star = d.value() # master solution
        if verbose:
            print(f"Iteration {i}, candidate costs: {d_star}")

        # solve subproblem
        sub_model.maximize(sum(x_0 * d_star))
        sub_model.solve()
        if verbose:
            print(f"  Is foil-based solution now optimal? {sum(d_star * x_d)} >=? {sum(d_star * x_0.value())}")
        if sum(d_star * x_d) >= sum(d_star * x_0.value()):
            return d_star # is optimal
        else:
            # add cutting plane to master
            master_model += [sum(d * x) >= sum(d * x_0.value())]
        i += 1

    raise ValueError("Master model is UNSAT!")
```

2. Integrated machine learning and solving

“CP as an oracle”

- Visual Sudoku (perception + solving)
- Preference learning in vehicle routing
- Decision-focussed learning (predict + optimize)

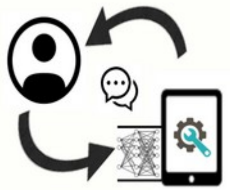
Why CPMpy?

“CP as an oracle”

European Research Council
Established by the European Commission

Consolidator grant (2021-2025)

“Conversational Human-aware Technology for Optimisation”



Still hiring post-docs!

CPMpy tutorial

Part 2: CP as an oracle, repeated solving

- Blocking constraints/solutions
- UNSAT core extraction
- Multiple solvers
- Implicit hitting set algorithms
- Cutting plane algorithms

CPMpy tutorial

Part 1: a CP modeling library in Python, based on numpy with direct solver access

Part 2: CP as an oracle, repeated solving

Part 3: peak behind the scenes

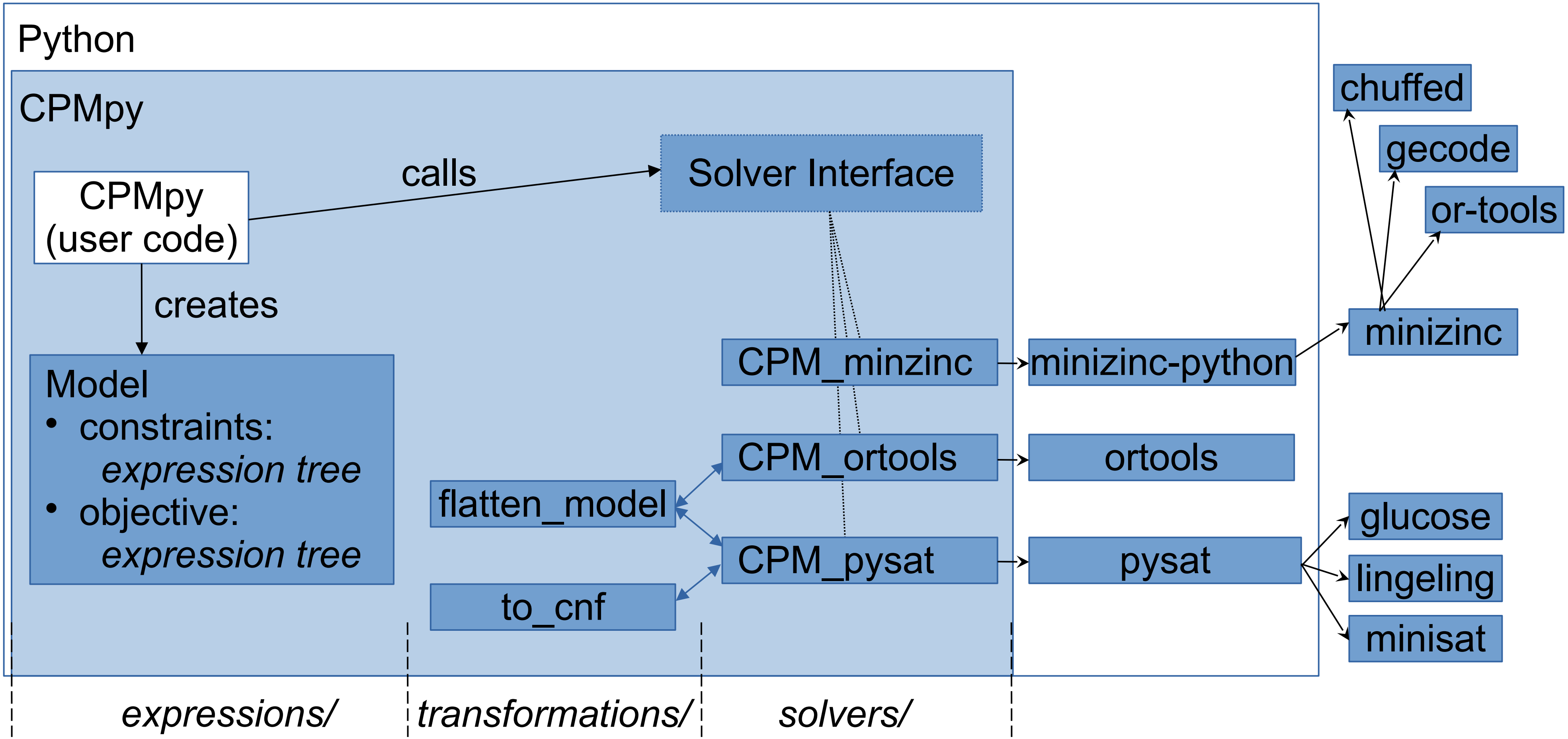
Design

Design principle:

Aim to be a thin layer on top of solver API

Central concept: CPMpy expression

Toolchain



Python
CPMpy
CPMpy (user code)
calls
Solver Interface
creates
Model
• constraints: expression tree
• objective: expression tree
CPM_minzinc
minizinc-python
minizinc
chuffed
gecode
or-tools
flatten_model
CPM_ortools
ortools
to_cnf
CPM_pysat
pysat
glucose
lingeling
minisat
expressions/
transformations/
solvers/

File structure

cpmy/

    __init__.py

    model.py

    expressions/

    transformations/

    solvers/

docs/

examples/

tests/

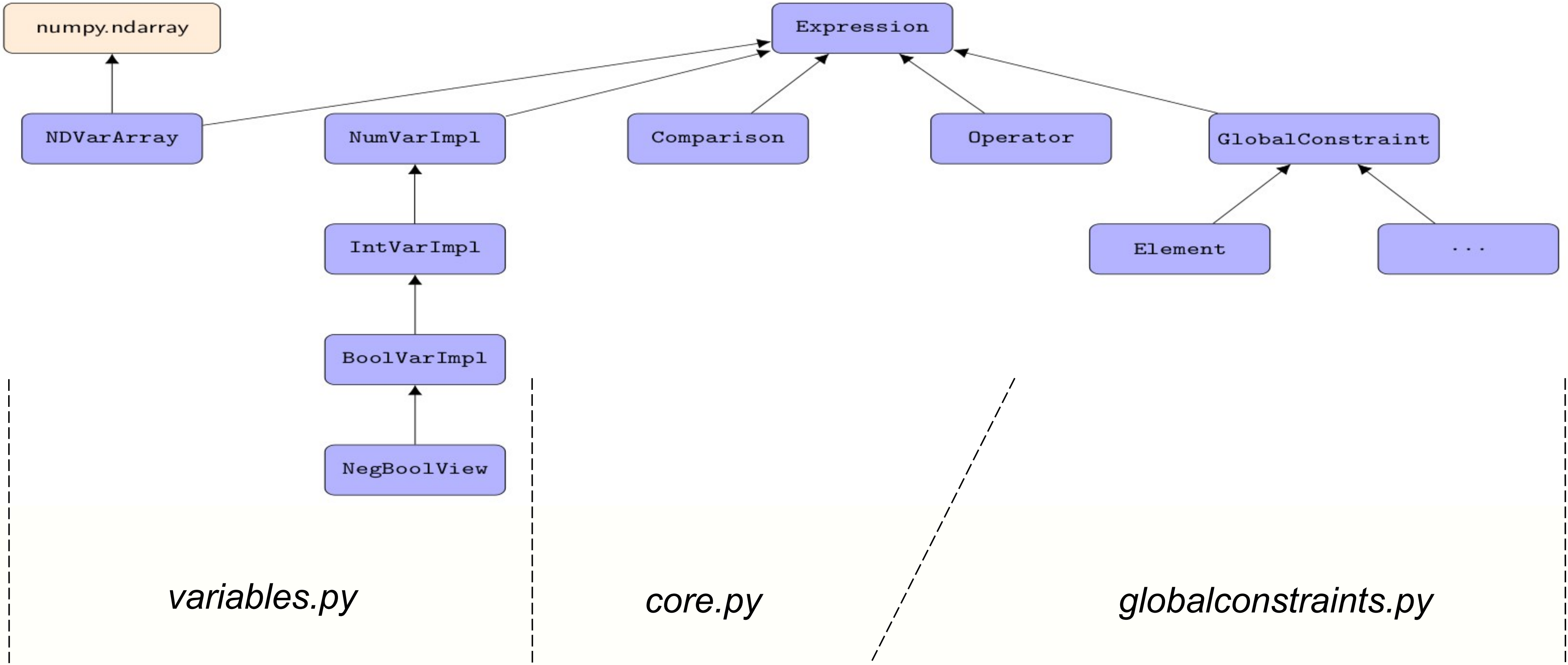
Generic container for expressions

All kinds of expression objects

Common methods to rewrite expressions

Classes that translate

Class Diagram of *expressions/*



Transformations

Key concept: ‘Flat Normal Form’

Like Negated Normal Form and
Conjunctive Normal Form

but for CP (basically: limited-nesting negated normal form)

In line with what other languages call ‘*flattening*’,
but as a normal form

Flat Normal Form

```
The three families of possible constraints are:

Base constraints: (no nesting)
------------------------------

    - Boolean variable
    - Boolean operators: and([Var]), or([Var]), xor([Var]) (CPMpy class 'Operator', is_bool())
    - Boolean impliciation: Var -> Var                      (CPMpy class 'Operator', is_bool())
    - Boolean equality: Var == Var                          (CPMpy class 'Comparison')
                        Var == Constant                     (CPMpy class 'Comparison')
    - Global constraint (Boolean): global([Var]*)           (CPMpy class 'GlobalConstraint', is_bool())

Comparison constraints: (up to one nesting on one side)
-------------------------------------------------------

    - Numeric equality:  Numexpr == Var                     (CPMpy class 'Comparison')
                         Numexpr == Constant                (CPMpy class 'Comparison')
    - Numeric disequality: Numexpr != Var                   (CPMpy class 'Comparison')
                           Numexpr != Constant              (CPMpy class 'Comparison')
    - Numeric inequality (>=,>,<,<=): Numexpr >=< Var       (CPMpy class 'Comparison')

    Numexpr:

        - Operator (non-Boolean) with all args Var/constant (examples: +,*,/,mod,wsum)
                                                            (CPMpy class 'Operator', not is_bool())
        - Global constraint (non-Boolean) (examples: Max,Min,Element)
                                                            (CPMpy class 'GlobalConstraint', not is_bool()))

    wsum: wsum([Const],[Var]) represents sum([Const]*[Var]) # TODO: not implemented yet

Reify/imply constraint: (up to two nestings on one side)
--------------------------------------------------------

    - Reification (double implication): Boolexpr == Var    (CPMpy class 'Comparison')
    - Implication: Boolexpr -> Var                          (CPMpy class 'Operator', is_bool())
                   Var -> Boolexpr                          (CPMpy class 'Operator', is_bool())

    Boolexpr:

        - Boolean operators: and([Var]), or([Var]), xor([Var]) (CPMpy class 'Operator', is_bool())
        - Boolean equality: Var == Var                          (CPMpy class 'Comparison')
        - Global constraint (Boolean): global([Var]*)           (CPMpy class 'GlobalConstraint', is_bool())
        - Comparison constraint (see above)                     (CPMpy class 'Comparison')

    Reification of a comparison is the most complex case as it can allow up to 3 levels of nesting in total, e.g.:

        - (wsum([1,2,3],[IV1,IV2,IV3]) > 5) == BV
        - (IV1 == IV2) == BV
        - (BV1 == BV2) == BV3
```

Solvers

We only interface to Python APIs
(unfortunately, no Common CP solver API : (

Key principle: solver can implement any subset of expressions!

Solvers can also choose to:

- Support assumptions or not
- Be incremental or not
- Expose own solver parameters

Currently:
- ortools
- pysat
- minizinc

Near future: gurobi, ExactSolver
Wishlist: Z3, Mistral2, Geas

CPMpy tutorial

Part 1: a CP modeling library in Python, based on numpy with direct solver access

Part 2: CP as an oracle, repeated solving

Part 3: peak behind the scenes (keep it light) master cpmpy / examples / Go to file Add file ···

JayMan91 example predopt: shutil to remove cpmpy dir (#72) f92883a 1 hour ago History

..		
advanced	examp	
visual_examples	try-ex	
README.md	updat	
bibd.py	rework	
bus_schedule.py	rework	
frietkot.py	examp	
jobshop.py	examp	
knapsack.py	ex/kna	
lpcp21_p1_frog.py	LP/CF	
mario.py	tests:	
minesweeper.py	examp	
npuzzle.py	npuzz	
nqueens.py	examp	
quickstart_sudoku.ipynb	examp	
send_more_money.py	rework	
sudoku.py	updat	
tsp.ipynb	examples: change tsp/vrp coordinates, tsp with circuit	3 months ago
tsp.py	tests: fix deprecation warnings	20 days ago
vrp.py	prefer exp.sum() over np.sum(exp)	18 days ago
who_killed_agatha.py	examples/who_killed_agatha: rework	3 months ago
wolf_goat_cabbage.py	Reworked WGC based on comments	9 days ago
zebra.py	examples: add zebra (integer)	3 months ago

master cpmpy / examples / advanced / Go to file Add file

JayMan91 example predopt: shutil to remove cpmpy dir (#72) f92883a 1 hour ago

..		
weights	lenet weights	4 d
VRP by learning from historical data.ipynb	Create VRP by learning from historical data.ipynb (#63)	9 ho
counterfactual_explain.py	Counterfactual example (#67)	12 ho
cp_explanations.py	Removing unused imports	5 d
diverse_solutions.py	example of diverse solutions	3 mor
hyperparameter_search.py	move param_combinations to cpmpy.solvers.utils	3 mor
marco_musmss_enumeration.py	musx/marco examples: polish printing	21 d
musx.py	musx: handle empty cases	18 d
ocus_explanations.py	unbreak	5 d
predict_plus_optimize.ipynb	example predopt: shutil to remove cpmpy dir (#72)	1 h
visual_sudoku.ipynb	update visual sudoku notebook	4 d

CPMpy tutorial

If you are a:

- CP user: give CPMpy a try, its fun

- Solver developer:

  we can make your solver easier to use,
  if you make a Python API (contact us)

Report issues and new examples on
https://github.com/CPMpy/cpmpy

Future Work

- More examples
- More advanced examples (paper reimplementations)
- More solvers (esp. incremental solvers!)

- towards Conversational, Human-Aware Technology for CP

Still hiring post-docs!

cpmpy.readthedocs.io/en/latest/

CPMpy
latest

Search docs

GETTING STARTED:

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

USER DOCUMENTATION:

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

API DOCUMENTATION:

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

» CPMpy: Constraint Programming and Modeling in Python

Edit on GitHub

CPMpy: Constraint Programming and Modeling in Python

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

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

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

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

Getting started:

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

User Documentation:

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

API documentation:

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

https://cpmpy.readthedocs.io/