CPMpy

Release 0.5

Tias Guns

Jun 28, 2022

GETTING STARTED:

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:

CHAPTER

ONE

GETTING STARTED WITH CONSTRAINT PROGRAMMING AND CPMPY

1.1 Constraint Programming

Many real-life decision problems involve searching over a large number of possible solutions to find one that satisfies all constraints and/or optimizes an objective function. For example in timetabling, scheduling, packing, routing and many more.

To decide if a problem is feasible or finding the best one amongst all the options is hard task to do by hand. And enumerating all possible solutions and simply checking whether they are good (generate-and-test) is usually infeasible in practice.

Instead, the paradigm of **constraint programming (CP)** allow you to:

1. Model the space of possible solutions through *decision variables*
2. Model relations between variables through *constraints* and an *objective function*
3. Have a state-of-the-art solver compute the answer efficiently.

So despite the word 'Programming' in Constraint Programming (since forever), as a user you only have to focus on *modeling* the problem, not on programming the search. This is the convenience and appeal of Constraint Programming.

1.2 Satisfaction versus Optimisation

A **constraint satisfaction problem (CSP)** consists of a set of variables and constraints stablishing relationships between them. Each variable has a finite of possible values (its domain). The goal is to assign values to the variables in its domains satisfying all the constraints.

A more general version, called **constraint optimization programming (C0P)**, finds amongst all the feasible solutions the one that optimizes some measure, called 'objective function'.

The state-of-the-art CP solvers can perform both very efficiently, so it is up to you to decide wether you have a satisfaction or an optimisation problem.

1.3 What is necessary to model a CP problem?

A typical CP problem is defined by the following elements:

Variables: Variables represents the decisions to be made. Depending on the decisions to be made variables can be *Boolean*, whenever a Yes or No decision is needed to be made, or *Integer*, whenever an integer number is necessary to represent a decision. In the first case, we say the **domain** of a Boolean variable is the set {True, False}. For integer variables we represent this as an interval of integer numbers, [a,b].

Constraints: Constraints are all the conditions that variables must satisfy. A set of values of the variables satisfying all the constraints is named a *feasible* solution. In CP, constraints can be boolean expressions, arithmetic operations or global constrains.

Moreover, if we want to model an constrained optimization problem we also need to specify an

Objective function: This is a function of the set of variables returning a real number. This metric is *maximized* or *minimized* over the set of all feasible solutions. An *optimal solution* is the one that satisfies all the constrains and returns the biggest value of the objective function (the smallest in case of minimization).

1.4 Example: cryptarithemtic

A cryptarithmetic puzzle is a mathematical challenge where the digits of some numbers are represented by letters (or symbols). Each letter represents a unique digit. The goal is to find the digits such that a given mathematical equation is verified.

For example, we aim to allocate to the letters S,E,N,D,M,O,R,Y a digit between 0 and 9, being all the letters allocated to a different digit and such that the expression:

SEND + MORE = MONEY

is satisfied. This problem lies into the setting of **constraint satisfaction problem (CSP)**. Here the variables are each letter S,E,N,D,M,O,R,Y and their domain is {0,1,2,...,9}. The constraints represents the fact that the values of the ltters need to sum up. And to be mathematically clean, the first letters can not be *0*.

1.5 Cryptarythmetic in CPMpy

First we need to import all the tools that we will need to create our CP model, namely numpy and our CPMpy library:

```
import numpy as np
from cpmpy import *
```

Secondly, as in every constraint programming model we need to define the decision variables:

```
s,e,n,d,m,o,r,y = intvar(0,9, shape=8)
```

This line indicates that we are creating 8 integer decision variables, s,e,n,d,m,o,r,y, and each will take a value between 0 and 9 (inclusive) in the solution. The *shape* argument informs the shape of the tensor (in this case, a vector of size 8, unpacked over the individual letters).

Thirdly, the constraints. We will immediately wrap them in a *Model()* object:

Constraints are included in the model as a list. First, we create a list to add the constraints. Then, we append an 'all different constraint' in a straightforward fashion. Finally, we add the constraint saying SEND + MORE = MONEY.

```
model = Model(
    AllDifferent(s,e,n,d,m,o,r,y),
    (    sum(   [s,e,n,d] * np.array([       1000, 100, 10, 1]) ) \
       + sum(   [m,o,r,e] * np.array([       1000, 100, 10, 1]) ) \
      == sum( [m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]) ) ),
    s > 0,
    m > 0,
)
```

The first line uses the *AllDifferent* global constraint. It is a CP primitive that will enforce that all variables get a different value. CP solvers have highly optimized procedures to enforce such constraints, hence the choice to model this with one *AllDifferent* global constraint rather then specifying that each pair of variables to have different values.

The second line (split over 3 lines) enforces the mathematical relation. Because CPMpy is based on the omnipresent numpy scientific library, you can perform products and other operators on combinations of CPMpy and NumPy arrays.

The last two lines enforce that the starting digits are not 0.

1.6 Solving a CPMpy model

Solving a model is as easy as calling *.solve()* on it, which will automatically search for a solver installed on the system, and make it solve the model:

```
model.solve()
```

The return value will be whether the model was satisfiable or not (True/False) in case of a satisfaction problem, and what the optimal value was in case of an optimisation problem.

The solution will be backpopulated in the decision variables used, and can be obtained by calling the *.value()* function on a decision variable. For example:

```
if model.solve():
    print("  S,E,N,D =   ", [x.value() for x in [s,e,n,d]])
    print("  M,O,R,E =   ", [x.value() for x in [m,o,r,e]])
    print("M,O,N,E,Y =", [x.value() for x in [m,o,n,e,y]])
else:
    print("No solution found")
```

And that is all there is to it...

1.7 Cryptarythmetic optimisation problem

So far we have considered a _satisfaction_ problem, where we needed to find any satisfying solution (it was unique, see *multiple_solutions* doc on how to find out).

We now consider the 'SEND + MOST = MONEY' problem, where we wish to maximize the number formed by the letters 'MONEY'.

We first model the constraints as before:

```
import numpy as np
from cpmpy import *
```

(continues on next page)

(continued from previous page)

```
s,e,n,d,m,o,t,y = intvar(0,9, shape=8)

model = Model(
    AllDifferent(s,e,n,d,m,o,t,y),
    (    sum(   [s,e,n,d] * np.array([       1000, 100, 10, 1]) ) \
       + sum(   [m,o,s,t] * np.array([       1000, 100, 10, 1]) ) \
      == sum( [m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]) ) ),
    s > 0,
    m > 0,
)
```

And now the objective function. Note that this just *states* that it is a maximisation problem, it does not yet compute the maximization.

```
model.maximize(sum( [m,o,n,e,y] * np.array([10000, 1000, 100, 10, 1]) ))
```

And then we solve and print! Now how *solve()* does not return True/False as for a satisfaction problem, but returns the objective's value.

```
model.solve()
print("  S,E,N,D =   ", [x.value() for x in [s,e,n,d]])
print("  M,O,S,T =   ", [x.value() for x in [m,o,s,t]])
print("M,O,N,E,Y =", [x.value() for x in [m,o,n,e,y]])
```

If you want to maximize the value of the word 'MOST', this is only requires changing the objective (you can overwrite objectives and resolve the same model without any problem):

```
model.maximize(sum( [m,o,s,t] * np.array([1000, 100, 10, 1]) ) )
model.solve()
print("  S,E,N,D =   ", [x.value() for x in [s,e,n,d]])
print("  M,O,S,T =   ", [x.value() for x in [m,o,s,t]])
print("M,O,N,E,Y =", [x.value() for x in [m,o,n,e,y]])
```

1.8 And much more

To get more familiar with these concepts, you can experiment with modeling and solving the sudoku puzzle problem in the following notebook.

And many more examples on scheduling, packing, routing and more in the examples folder.

1.9 References

To learn more about theory and practice of constraint programming you may want to check some of these references:

1. Rossi, F., Van Beek, P., & Walsh, T. (Eds.). (2006). Handbook of constraint programming. Elsevier.
2. Apt, K. (2003). Principles of constraint programming. Cambridge university press.

CHAPTER

TWO

INSTALLATION INSTRUCTIONS

CPMpy requires Python `3.6` or newer. The package is available on PyPI.

The easiest way is to install using the 'pip' command line package manager. In a terminal, run:

```
$ pip install cpmpy
```

This will automatically also install the default 'ortools' solver.

If the previous command fails to execute, it may be due to the permission to install the package globally Python packages. If so, you can install it for your user only with:

```
$ pip install cpmpy --user
```

CPMpy has regular small releases with updates and improvements, so it is a good habbit to regularly update, as follows:

```
$ pip install -U cpmpy
```

2.1 Installing from a git repository

If you want the very latest, or perhaps from an in-development branch, you can install directly from github as follows:

```
$ pip install git+https://github.com/cpmpy/cpmpy@master
```

(change 'master' to any other branch or commit hash)

2.2 Installing a local copy

If you are developing CPMpy locally, you can run scripts from in the repository folder, and it will use the cpmpy/ folder as package instead of any installed one.

However, if you want to test some local changes to CPMpy that can only be tested by installing CPMpy, you can do that as follows from the repository folder:

```
$ pip install .
```

CHAPTER

THREE

MODELING

CPMpy is a library for modeling and solving constrained satisfaction and optimisation problems (CSPs and COPs in the AI literature).

A constraint model consists of 3 key parts:

- *Decision variables* with their domain (allowed values)
- *Constraints* over decision variables
- Optionally an *objective function*

```
from cpmpy import *
m = Model()

# Variables
b = boolvar(name="b")
x = intvar(1,10, shape=3, name="x")

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

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

print(m)
print(m.solve(), x.value(), b.value())
```

See https://github.com/CPMpy/cpmpy/blob/master/examples/quickstart_sudoku.ipynb for a more realistic step-by-step example.

3.1 Variables

CPMpy supports discrete decision variables. All variables are numpy arrays, so creating 1 of them or an array/tensor of them is similar:

- Boolean variables: `boolvar(shape=1, name=None)`
- Integer variables: `intvar(lb, ub, shape=1, name=None)`

See the API documentation on variables for more information.

3.2 Constraints

Constraints have to be *added* to a model, which is done with the += operator, e.g. `model += constraint`. You can also add lists of constraints and other nested expressions.

Using Python's built-in comparison operators `==, !=, <, <=, >, >=` and logical operators `&, |, ~, ^` on CPMpy variables will automatically create constraint expressions that can be added to a model.

You can also use the built-in arithmetic operators `+, -, *, //, %` and we overwrite the built-in `abs, sum, min, max, all, any` functions so that you can use them in the construction of expressions.

CP languages like CPMpy also offer what is called **global constraints**. Convenient expressions that capture part of the problem structure and that the solvers can typically use efficiently too. A non-exhaustive list of global constraints is: `AllDifferent(), AllEqual(), Circuit(), Table(), Element()`.

See the API documentation on expressions for more information.

3.3 Objective function

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, this function needs to be minimized or maximized.

Any CPMpy expression can be added as objective function. Solvers are especially good in optimizing linear functions or the minimum/maximum of a set of expressions. Other (non-linear) expressions are supported too, just give it a try.

3.4 Other uses of the `Model()` object

The `Model()` object has a number of other helpful functions, such as `status()` to print the status of the last `solve()` call, `solveAll()` to find all solutions, `to_file()` to store the model (you can print a model too, for debugging) and `copy` for creating a copy.

See the API documentation on Model for more information.

CHAPTER

FOUR

SOLVERS

CPMpy can be used as a declarative modeling language: you create a `Model()`, add constraints and call `solve()` on it.

The default solver is ortools CP-SAT, an award winning constraint solver. But CPMpy supports multiple other solvers: a MIP solver (gurobi), SAT solvers (those in PySAT) and any CP solver supported by the text-based MiniZinc language.

See the list of solvers known by CPMpy with:

```
SolverLookup.solvernames()
```

Note that many require additional packages to be installed. For example, try `SolverLookup.get("gurobi")` to see if the commercial gurobi solver is available on your system. See the API documentation of the solver for installation instructions.

You can specify a solvername when calling `solve()` on a model:

```
from cpmpy import *
x = intvar(0,10, shape=3)
m = Model()
m += sum(x) <= 5
# use named solver
m.solve(solver="ortools")
```

In this case, a model is a **lazy container**. It simply stores the constraints. Only when `solve()` is called will it instantiate a solver, and send the entire model to it at once. The last line above is equivalent to:

```
s = Solverlookup.get("ortools", m)
s.solve()
```

4.1 Model versus solver interface

Solver interfaces allow more than the generic model interface, because, well, they can support solver-specific features. Such as solver-specific parameters, passing a previous solution to start from, incremental solving, unsat core extraction, solver-specific callbacks etc.

Importantly, the solver interface supports the same functions as the `Model()` object (for adding constraints, an objective, solve, solveAll, status, ...). So if you want to make use of some features of a solver, simply replace `m = Model()` by `m = SolverLookup.get("your-preferred-solvername")` and your code remains valid. Below, we replace `m` by `s` for readability.

```
from cpmpy import *
x = intvar(0,10, shape=3)
s = SolverLookup.get("ortools")
s += sum(x) <= 5
# we are operating on the ortools interface here
s.solve()
```

4.2 Setting solver parameters

Now lets use our solver-specific powers: ortools has a parameter _log_search*progress* that make it show information during solving for example:

```
# we are operating on the ortools interface here
s.solve(log_search_progress=True)
```

Modern CP-solvers support a variety of hyperparameters. (OR-tools parameters for example). Using the solver interface, any solver parameter can be passed using the `.solve()` call. These parameters will then be posted to the native solver object before solving the model.

```
s.solve(cp_model_probing_level = 2,
        linearization_level = 0,
        symmetry_level = 1)
```

See the API documentation of the solvers for information and links on the parameters supported. See our documentation page on solver parameters if you want to tune your hyperparameters automatically.

4.3 Using solver-specific CPMpy functions

We sometimes add solver-specific features to the CPMpy interface, for convenient access. Two examples of this are `solution_hint()` and `get_core()` which is supported by the OrTools and PySAT solvers and interfaces. Other solvers work very different and do not have these concepts.

`solution_hint()` tells the solver that it could use these variable-values first during search, e.g. typically from a previous solution:

```
from cpmpy import *
x = intvar(0,10, shape=3)
s = SolverLookup.get("ortools")
s += sum(x) <= 5
# we are operating on a ortools' interface here
s.solution_hint(x, [1,2,3])
s.solve()
print(x.value())
```

`get_core()` asks the solver for an unsatisfiable core, in case a solution did not exist and assumption variables were used. See the documentation on Unsat core extraction.

See the API documentation of the solvers to learn about their special functions.

4.4 Incremental solving

It is important to realize that a CPMpy solver interface is *eager*. That means that when a CPMpy constraint is added to a solver object, CPMpy *immediately* translates it and posts the constraints to the underlying solver.

This has two potential benefits for incremental solving, whereby you add more constraints and variables inbetween solve calls:

1) CPMpy only translates and posts each constraint once, even if the model is solved multiple times; and
2) if the solver itself is incremental then it can reuse any information from call to call, as the state of the native solver object is kept between solver calls and can therefore rely on information derived during a previous `solve` call.

```
gs = SolverLookup.get("gurobi")

gs += sum(ivar) <= 5
gs.solve()

gs += sum(ivar) == 3
# underlying solver instance is reused, only the new constraint is added to it
# gurobi can start looking for solutions at previous solution
gs.solve()
```

Technical note: ortools its model representation is incremental but its solving itself is not (yet?). Gurobi and the PySAT solvers are fully incremental. The text-based MiniZinc language is not incremental.

4.5 Native solver access and constraints

Another benefit of using a solver interface directly is access to low level solver features not implemented in CPMpy. The solver interface implemented by CPMpy encapsulates the native solver object and allows users to access these objects directly.

That means that you can mix posting CPMpy expressions as constraints, and posting **solver-specific global constraints** directly.

To get you started, the following simple model:

```
ffrom cpmpy import *
x = intvar(0,10, shape=3)
s = SolverLookup.get("ortools")

s += sum(x) > 10
s += AllDifferent(x)
s += x[1] == 5

s.solve()
print(x.value())
```

can equivalently be written by posting the native `AddAllDifferent()` directly on the underlying ortools object:

```
from cpmpy import *
x = intvar(0,10, shape=3)
s = SolverLookup.get("ortools")
```

(continues on next page)

(continued from previous page)

```
s += sum(x) > 10
s.ort_model.AddAllDifferent(s.solver_vars(x))
s += x[1] == 5

s.solve()
print(x.value())
```

observe how we first map the CPMpy variables to native variables by calling `s.solver_vars()`, and then give these to the native solver API directly. This is in fact what happens behind the scenes when posting a constraint.

CHAPTER

FIVE

OBTAINING MULTIPLE SOLUTIONS

CPMpy models and solvers support the `solveAll()` function. It efficiently computes all solutions and optionally displays them. Alternatively, you can manually add blocking clauses as explained in the second half of this page.

When using `solveAll()`, a solver will use an optimized native implementation behind the scenes when that exists.

It has two special named optional arguments:

- *display*=...: accepts a CPMpy expression, a list of CPMpy expressions or a callback function that will be called on every solution found (default: None)
- *solution_limit*=...: stop after this many solutions (default: None)

It also accepts named argument `time_limit=...` and any other keyword argument is passed on to the solver just like `solve()` does.

It returns the number of solutions found.

5.1 `solveAll()` examples

In the following examples, we assume:

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

Just return the number of solutions (here: 6)

```
n = m.solveAll()
print("Nr of solutions:", n)
```

With a solution limit: e.g. find up to 2 solutions

```
n = m.solveAll(solution_limit=2)
print("Nr of solutions:", n)
```

Find all solutions, and print the value of `x` for each solution found.

```
n = m.solveAll(display=x)
```

`display` Can also take lists of arbitrary CPMpy expressions:

```
n = m.solveAll(display=[x,sum(x)])
```

Perhaps most powerful is the use of **callbacks**, which allows for rich printing, solution storing, dynamic stopping and more. You can use any variable name from the outer scope here (it is a closure). That does mean that you have to call `var.value()` each time to get the value(s) of this particular solution.

Rich printing with a callback function:

```
def myprint():
    xval = x.value()
    print(f"x={xval}, sum(x)={sum(xval)}")
n = m.solveAll(display=myprint) # callback function without brackets
```

Also callback with an anonymous lambda function possible:

```
n = m.solveAll(display=lambda: print(f"x={x.value()} sum(x)={sum(x.value())}")
```

A callback is also the (only) way to go if you want to store information about all the found solutions (only recommended for small numbers of solutions).

```
solutions = []
def collect():
    print(x.value())
    solutions.append(list(x.value()))
n = m.solveAll(display=collect, solution_limit=1000) # callback function without brackets
print(f"Stored {len(solutions)} solutions.")
```

5.2 Solution enumeration with blocking clauses

The MiniSearch[1] paper promoted a small, high-level domain-specific language for specifying the search for multiple solutions with blocking clauses.

This approach makes use of the incremental nature of the solver interfaces. It is hence much more efficient (less overhead) to do this on a solver object rather then a generic model object.

Here is an example of standard solution enumeration, note that this will be much slower than `solveAll()`.

```
from cpmpy import *

x = intvar(0,3, shape=2)
m = Model(x[0] > x[1])
s = SolverLookup.get("ortools", m) # faster on a solver interface directly

while s.solve():
    print(x.value())
    # block this solution from being valid
    s += ~all(x == x.value())
```

In case of multiple variables you should put them in one long python-native list, as such:

```
x = intvar(0,3, shape=2)
b = boolvar()
m = Model(b.implies(x[0] > x[1]))
s = SolverLookup.get("ortools", m) # faster on a solver interface directly
```

(continues on next page)

(continued from previous page)

```
while s.solve():
    print(x.value(), b.value())
    allvars = list(x)+[b]
    # block this solution from being valid
    s += ~all(v == v.value() for v in allvars)
```

5.3 Diverse solution search

A better, more complex example of repeated solving is when searching for diverse solutions.

The goal is to iteratively find solutions that are as diverse as possible with the previous solutions. Many definitions of diversity between solutions exist. We can for example measure the difference between two solutions with the Hamming distance (comparing the number of different values) or the Euclidian distance (compare the absolute difference in value for the variables).

Here is the example code for enumerating K diverse solutions with Hamming distance, which overwrites the objective function in each iteration:

```
# Diverse solutions, Hamming distance (inequality)
x = boolvar(shape=6)
m = Model(sum(x) == 2)
s = SolverLookup.get("ortools", m) # faster on a solver interface directly

K = 3
store = []
while len(store) < 3 and s.solve():
    print(len(store), ":", x.value())
    store.append(x.value())
    # Hamming dist: nr of different elements
    s.maximize(sum([sum(x != sol) for sol in store]))
```

As a fun fact, observe how `x != sol` works, even though one is a vector of Boolean variables and sol is Numpy array. However, both have the same length, so this is automatically translated into a pairwise comparison constraint by CPMpy. These numpy-style vectorized operations mean we have to write much less loops while modelling.

To use the Euclidian distance, only the last line needs to be changed. We again use vectorized operations and obtain succinct models. The creation of intermediate variables (with appropriate domains) is done by CPMpy behind the scenes.

```
# Euclidian distance: absolute difference in value
s.maximize(sum([sum( abs(np.add(x, -sol)) ) for sol in store]))
```

5.4 Mixing native callbacks with CPMpy

CPMpy passes arguments to `solve()` directly to the underlying solver object, so you can actually define your own native callbacks and pass them to the solve call.

The following is an example of that, which is actually how the native `solveAll()` for ortools is implemented. You could give it your own custom implemented callback `cb` too.

```
from cpmpy import *
from cpmpy.solvers import CPM_ortools
from cpmpy.solvers.ortools import OrtSolutionPrinter

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

s = SolverLookup.get('ortools', m)
cb = OrtSolutionPrinter()
s.solve(enumerate_all_solutions=True, solution_callback=cb)
print("Nr of solutions:",cb.solution_count())
```

CHAPTER

SIX

HOW TO DEBUG

You get an error, or no error, but also no (correct) solution… Annoying, you have a bug.

The bug can be situated in one of three layers:

- your problem specification
- the CPMpy library
- the solver

coincidentally, they are ordered from most likely to least likely. So let's start at the bottom.

If you don't have a bug yet, but are curious, here is some general advise 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, try to comment out all the constraints and then activate them again one by one to test which constraint is the culprit.
- 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.

6.1 Debugging the solver

If you get an error and have difficulty understanding it, try searching on the internet if other users have had the same.

If you don't find it, or if the solver runs fine and without error, but you don't get the answer you expect; then try swapping out the solver for another solver and see what gives…

Replace `model.solve()` by `model.solve(solver='minizinc')` for example. You do need to install MiniZinc and minizinc-python first though.

Either you have the same output, and it is not the solver's fault, or you have a different output and you actually found one of these rare solver bugs. Report on the bugtracker of the solver, or on the CPMpy github page where we will help you file a bug 'upstream' (or maybe even work around it in CPMpy).

6.2 Debugging a modeling error

You get an error when you create an expression? Then you are probably writing it wrongly. Check the documentation and the running examples for similar examples of what you wish to express.

Here are a few quirks in Python/CPMpy:

- when using `&` and `|`, make sure to always put the subexpressions in brackets. E.g. `(x == 1) & (y == 0)` instead of `x == 1 & y == 0`. The latter wont work, because Python will unfortunately think you meant `x == (1 & (y == 0))`.
- you can write `vars[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 = cpm_array(non_var_list)` first, or write `Element(non_var_list, a_var)` instead.
- only write `sum(v)` on lists, don't write it if `v` is a matrix or tensor, as you will get a list in response. Instead, use NumPy's `v.sum()` instead.

Try printing the expression `print(e)` or subexpressions, and check that the output matches what you wish to express. Decompose the expression and try printing the individual components and their piecewice composition to see what works and when it starts to break.

If you don't find it, report it on the CPMpy github Issues page and we'll help you (and maybe even extend the above list of quirks).

6.3 Debugging a `solve()` error

You get an error either from CPMpy (e.g. the flattening, or the solver interface) or the solver itself is saying the model is invalid. This may be because you have modelled something impossible, or because you have a corner case that CPMpy does not yet capture.

If you have a model that fails in this way, try the following code snippet to see what constraint is causing the error:

```
model = ... # your code, a `Model()`
`
for c in model.constraints:
    print("Trying",c)
    Model(c).solve()
```

The last constraint printed before the exception is the curlpit... Please report on Github. We want to catch corner cases in CPMpy, even if it is a solver limitation, so please report on the CPMpy github Issues page.

Or maybe, you got one of CPMpy's NotImplementedErrors. Share your use case with us on Github and we will implement it. Or implemented it yourself first, that is also very welcome ;)

6.4 Debugging an UNSATisfiable model

First, print the model:

`print(model)` and check that the output matches what you want to express. Do you see anything unusual? Start there, see why the expression is now what you intended to express, as described in 'Debugging a modeling error'.

If that does not help, try printing the 'flat normal form' of the model, which also shows the intermediate variables that are automatically created by CPMpy:

```
from cpmpy.transformations.flatten_model import flatten_constraint, flatten_model
print(flatten_model(model))
```

Note that you can also print individual flattened expressions with `print(flatten_constraint(expression))` which helps to zoom in on the curlpit.

If you want to know about the variable domains as well, to see whether something is wrong there, you can do so as follows:

```
from cpmpy.transformations.flatten_model import flatten_constraint, flatten_model
from cpmpy.transformations.get_variables import print_variables
mf = flatten_model(model)
print_variables(mf)
print(mf)`
```

6.4.1 Automatically minimising the UNSAT program

If the above is unwieldy because your constraint problem is too large, then consider automatically reducing it to its 'UNSAT core'.

You can download our 'musx.py' advanced example. If you put it in the same folder as your code, then you can use it as follows:

```
from musx import musx

x,y,z = boolvar(3)
model = Model(
    x,
    ~x,
    x|y,
    z.implies(x)
    )

unsat_cons = musx(model.constraints)
model2 = Model(unsat_cons)
```

With this smaller program, repeat the visual inspection steps above.

6.5 Debugging a satisfiable model, that does not contain an expected solution

We will ignore the (possible) objective function here and focus on the feasibility part. Actualy, in case of an optimisation problem where you know a certain value is attainable, you can add `objective == known_value` as constraint and proceed similarly.

Add the solution that you know should be a feasible solution as a constraint: `model.add( (x == 1) & (y == 2) & (z == 3) ) # yes, brackets around each!`

You now have an UNSAT program! That means you can follow the steps in 'Automatically minimising the UNSAT program' above to better understand it.

6.6 Debugging a satisfiable model, which returns an impossible solution

This one is most annoying… Double check the printing of the model for oddities, also visually inspect the flat model. Try enumerating all solutions and look for an unwanted pattern in the solutions. Try a different solver.

Try generating an explanation sequence for the solution… this requires a satisfaction problem, so remove the objective function or add a constraint that constraints the objective function to the value attained by the impossible solution.

As to generating the explanation sequence, we will add this as a demo soon…

CHAPTER

SEVEN

SETTING SOLVER PARAMETERS AND HYPERPARAMETER SEARCH

7.1 Calling a solver by name

You can see the list of available solvers (and subsolvers) as follows:

```
from cpmpy import *

print(SolverLookup.solvernames())
```

On my system, with pysat and minizinc installed, this gives `['ortools', 'minizinc', 'minizinc:chuffed', 'minizinc:coin-bc', ..., 'pysat:minicard', 'pysat:minisat22', 'pysat:minisat-gh']

You can use any of these solvers by passing its name to the `Model.solve()` parameter 'solver' as such:

```
a,b = boolvar(2)
Model(a|b).solve(solver='minizinc:chuffed')
```

7.2 Direct solver access

CPMpy also offers direct access to its solver API, as well as to the underlying native solver API. For most cases, including setting solver parameters, access to CPMpy's solver API will be sufficient.

In the following, we will use the or-tools CP-SAT solver. The corresponding CPMpy class is `CPM_ortools` and can be included as follows:

```
from cpmpy.solvers import CPM_ortools
```

The same principles will apply to the other solver interfaces too.

7.3 Setting solver parameters

or-tools has many solver parameters, documented here.

CPMpy's interface to ortools accepts keyword arguments to `solve()`, and will set the corresponding or-tools parameters if the name matches. We documented some of the frequent once in our CPM_ortools API.

For example, with `model` a CPMpy Model(), you can do the following to make or-tools use 8 parallel cores and print search progress:

```
from cpmpy import *
from cpmpy.solvers import CPM_ortools

s = CPM_ortools(model)
s.solve(num_search_workers=8, log_search_progress=True)
```

7.4 Hyperparameter search across different parameters

Because CPMpy offers programmatic access to the solver API, hyperparameter search can be straightforwardly done with little overhead between the calls.

The cpmpy.solvers module has a helper function `param_combinations` that generates all parameter combinations of an input, which can then be looped over.

The example is in examples/advanced/hyperparameter_search.py, the key part is:

```
from cpmpy.solvers import CPM_ortools, param_combinations

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

for params in param_combinations(all_params):
    s = CPM_ortools(model)
    s.solve(**params)
    print(s.status().runtime, "seconds for config", params)
```

CHAPTER

EIGHT

UNSAT CORE EXTRACTION WITH ASSUMPTION VARIABLES

When a model is unsatisfiable, it can be desirable to get a better idea of which Boolean variables make it unsatisfiable. Commonly, these Boolean variables are 'switches' that turn constraints on, hence such Boolean variables can be used to get a better idea of which *constraints* make the model unsatisfiable.

In the SATisfiability literature, the Boolean variables of interests are called *assumption* variables and the solver will assume they are true. The subset of these variables that, when true, make the model unsatisfiable is called an unsatisfiable *core*.

Lazy Clause Generation solvers, like or-tools, are built on SAT solvers and hence can inherit the ability to define assumption variables and extract an unsatisfiable core.

Since version 8.2, or-tools supports declaring assumption variables, and extracting an unsat core. We also implement this functionality in CPMpy, using PySAT-like `s.solve(assumptions=[...])` and `s.get_core()`:

```
from cpmpy import *
from cpmpy.solvers import CPM_ortools

bv = boolvar(shape=3)
iv = intvar(0,9, shape=3)

# circular 'bigger then', UNSAT
m = Model(
    bv[0].implies(iv[0] > iv[1]),
    bv[1].implies(iv[1] > iv[2]),
    bv[2].implies(iv[2] > iv[0])
)

s = CPM_ortools(m)
print(s.solve(assumptions=bv))
print(s.status())
print("core:", s.get_core())
print(bv.value())
```

This opens the door to more advanced use cases, such as Minimal Unsatisfiable Subsets and QuickXplain-like tools to help debugging. We welcome any examples or additions that use CPMpy in this way!! Here is one example: the MARCO algorithm for enumerating all MUS/MSSes. Here is another: a deletion-based MUS algorithm, made to also work for non-reifable (global) constraints. Also useful to debug unsatisfiable models!

One final caveat is that the or-tools Python interface is by design *stateless*. That means that, unlike in PySAT, calling `s.solve(assumptions=bv)` twice for a different `bv` array does NOT REUSE anything from the previous run: no warm-starting, no learnt clauses that are kept, no incrementality, so there will be some pre-processing overhead. If you know of another CP solver with a (Python) assumption interface that is incremental, let us know!!

A final-final note is that you can manually warm-start or-tools with a previously found solution with s.solution_hint(); see also the MARCO code linked above.

CHAPTER

NINE

ADDING A NEW SOLVER

Any solver that has a Python interface can be added as a solver to CPMpy. See the bottom of this page for tips in case the/your solver does not have a Python interface yet.

To add your solver to CPMpy, you should copy cpmpy/solvers/TEMPLATE.py directory, rename it to your solver name and start filling in the template. You can also look at how it is done for other solvers, they all follow the template.

Implementing the template consists of the following parts:

- `supported()` where you check if the solver package is installed. Never include the solver python package at the top-level of the file, CPMpy has to work even if a user did not install your solver package.
- `__init__()` where you initialize the underlying solver object
- `solver_var()` where you create new solver variables and map them CPMpy decision variables
- `solve()` where you call the solver, get the status and runtime, and reverse-map the variable values after solving
- `objective()` if your solver supports optimisation
- `__add__()` where you call the necessary transformations to transform CPMpy expressions to those that the solver supports
- `_post_constraint()` where you directly map the CPMpy expressions that the solver supports, to API function calls on the underlying solver
- `solveAll()` optionally, if the solver natively supports solution enumeration

9.1 Transformations and posting constraints

CPMpy is designed to separate 'transforming' constraints as much as possible from 'posting' constraints.

For example, a SAT solver only accepts clauses (disjunctions) over Boolean variables as constraints. So, its `_post_constraint()` method should just consists of reading in a CPMpy 'or' expression over decision variables, for which it then calls the solver to create such a clause. All other constraints may not be directly supported by the solver, and can hence be rejected.

What remains is the difficult part of mapping an arbitrary CPMpy expression to CPMpy 'or' expressions. This is exactly the task of a constraint modelling language like CPMpy, and we implement it through multiple independent **transformation functions** in the `cpmpy/transformations/` directory. For any solver you wish to add, chances are that most of the transformations you need are already implemented. If not, read on.

9.2 Stateless transformation functions

CPMpy solver interfaces are *eager*, meaning that any CPMpy expression given to it (through `__add__()`) is immediately transformed and posted to the solver. That also allows it to be *incremental*, meaning that you can post some constraints, call `solve()` post some more constraints and solve again. If the underlying solver is also incremental, it will reuse knowledge of the previous solve call to speed up this solve call.

The way that CPMpy succeeds to be an incremental modeling language, is by making all transformation functions *stateless*. Every transformation function is a python *function* that maps a (list of) CPMpy expressions to (a list of) equivalent CPMpy expressions. Transformations are not classes, they do not store state, they do not know (or care) what model a constraint belongs too. They take expressions as input and compute expressions as output. That means they can be called over and over again, and chained in any combination or order.

That also makes them modular, and any solver can use any combination of transformations that it needs. We continue to add and improve the transformations, and we are happy to discuss transformations you are missing, or variants of existing transformations that can be refined.

Most transformations do not need any state, they just do a bit of rewriting. Some transformations do, for example in the case of common subexpression elimination. In that case, the solver interface (you who are reading this), should store a dictionary in your solver interface class, and pass that as (optional) argument to the transformation function. The transformation function will read and write to that dictionary as it needs, while still remaining stateless on its own. Each transformation function documents when it supports an optional state dictionary, see all available transformations in `cpmpy/transformations/`.

9.3 What is a good Python interface for a solver?

A *light-weight, functional* API is what is most convenient from the CPMpy perspective, as well as in terms of setting up the Python-C++ bindings (or C, or whatever language the solver is written in).

With **functional** we mean that the API interface is for example a single class that has functions for adding variables, constraints and solve actions that it supports.

What we mean with **light-weight** is that it has none or few custom data-structures exposed at the Python level. That means that the arguments and return types of the API consist mostly of standard integers/strings/lists.

Here is a fictive pseudo-code of such an API, which is heavily inspired on the Ortools CP-SAT interface:

```
class SolverX {
    private Smth real_solver;

    // constructor
    void SolverX() {
        real_solver = ...; // internal solver object, not exported to Python
    }

    // managing variables
    str addBoolVar(str name); // returns unique variable ID (can also be a light-weight␣
↪struct)
    str addIntVar(int lb, int ub, str name): // returns unique variable ID

    int getVarValue(str varID); // obtaining the value of a variable after solve

    // adding constraints
    void postAnd(vector<str> varIDs);
```

(continues on next page)

(continued from previous page)

```
    void postAndImplied(str boolID, vector<str> varIDs); // bool implies and(vars)
    void postOr(vector<str> varIDs);
    void postOrImplied(str boolID, vector<str> varIDs);
    void postAllDifferent(vector<str> varIDs);
    void postSum(vector<str> varIds, str Operator, str varID);
    void postSum(vector<str> varIds, str Operator, int const);
    // I think or-tools actually creates a map (unique ID) for both variables and␣
↪constants, so they can be used in the same expression
    void postWeightedSum(vector<str> varIds, vector<int> weights, str Operator, str␣
↪varID);
    ...

    // adding objective
    void setObjective(str varID, bool is_minimize);
    void setObjectiveSum(vector<str> varID, bool is_minimize);
    void setObjectiveWeightedSum(vector<str> varID, vector<int> weights, bool is_
↪minimize);
    ...

    // solving
    int solve(bool param1, int param2, str param3, ...); // return-value represents␣
↪return state (opt, sat, unsat, error, ...)
    ...
}
```

If you have such a C++ API, then there exist automatic python packages that can make Python bindings, such as CPPYY.

We have not done this ourselves yet, so get in touch to share your experience and advice!

CHAPTER

TEN

EXPRESSIONS (CPMPY.EXPRESSIONS)

All forms of expression objects that allow you to specify constraints and objectives over variables

10.1 List of submodules

variables	Integer and Boolean decision variables (as n-dimensional numpy objects)
core	The *Expression* superclass and common subclasses *Expression* and *Operator*.
globalconstraints	Global constraints conveniently express non-primitive constraints.
python_builtins	Overwrites a number of python built-ins, so that they work over variables as expected.
utils	Internal utilities for expression handling.

CHAPTER

ELEVEN

MODEL (CPMPY.MODEL)

The *Model* class is a lazy container for constraints and an objective function.

It is lazy in that it only stores the constraints and objective that are added to it. Processing only starts when solve() is called, and this does not modify the constraints or objective stored in the model.

A model can be solved multiple times, and constraints can be added to it inbetween solve calls.

See the examples for basic usage, which involves:

- creation, e.g. m = Model(cons, minimize=obj)
- solving, e.g. m.solve()
- optionally, checking status/runtime, e.g. m.status()

11.1 List of classes

Model	CPMpy Model object, contains the constraint and objective expressions

class cpmpy.model.**Model**(**args*, *minimize=None*, *maximize=None*)

CPMpy Model object, contains the constraint and objective expressions

copy()

Makes a shallow copy of the model. Constraints and variables are shared among the original and copied model.

deepcopy(*memodict={}*)

Deep copies a the model to a new instance. :return: an object of :class: 'Model' with equivalent constraints as the current model. There are no shared variables/constraints between the original model and its copied version.

static from_file(*fname*)

Reads a Model instance from a binary pickled file

Returns
an object of :class: *Model*

maximize(*expr*)

Maximize the given objective function

maximize() can be called multiple times, only the last one is stored

`minimize`(*expr*)

Minimize the given objective function

minimize() can be called multiple times, only the last one is stored

`objective`(*expr*, *minimize*)

Post the given expression to the solver as objective to minimize/maximize

- expr: Expression, the CPMpy expression that represents the objective function
- minimize: Bool, whether it is a minimization problem (True) or maximization problem (False)

'objective()' can be called multiple times, only the last one is stored

`objective_value`()

Returns the value of the objective function of the latste solver run on this model

Returns
an integer or 'None' if it is not run, or a satisfaction problem

`solve`(*solver=None*, *time_limit=None*)

Send the model to a solver and get the result

Parameters

- **`solver`** – name of a solver to use. Run SolverLookup.solvernames() to find out the valid solver names on your system. (default: None = first available solver)
- **`time_limit`** (*`int or float`*) – optional, time limit in seconds

Returns
Bool: the computed output: - True if a solution is found (not necessarily optimal, e.g. could be after timeout) - False if no solution is found

`solveAll`(*solver=None*, *display=None*, *time_limit=None*, *solution_limit=None*)

Compute all solutions and optionally display the solutions.

Delegated to the solver, who might implement this efficiently

Arguments:

- **display: either a list of CPMpy expressions, OR a callback function, called with the variables after value-mapping**
  default/None: nothing displayed
- solution_limit: stop after this many solutions (default: None)

Returns: number of solutions found

`status`()

Returns the status of the latest solver run on this model

Status information includes exit status (optimality) and runtime.

Returns
an object of `SolverStatus`

`to_file`(*fname*)

Serializes this model to a .pickle format

Param
fname: Filename of the resulting serialized model

CHAPTER

TWELVE

SOLVER INTERFACES (CPMPY.SOLVERS)

CPMpy interfaces to (the Python API interface of) solvers

Solvers typically use some of the generic transformations in *transformations* as well as specific reformulations to map the CPMpy expression to the solver's Python API

12.1 List of submodules

ortools	Interface to ortools' CP-SAT Python API
pysat	Interface to PySAT's API
gurobi	Interface to the python 'gurobi' package
pysdd	Interface to PySDD's API
utils	Utilities for handling solvers

12.2 List of classes

CPM_ortools	Interface to the python 'ortools' CP-SAT API
CPM_pysat	Interface to PySAT's API
CPM_gurobi	Interface to Gurobi's API
CPM_pysdd	Interface to pysdd's API

12.3 List of functions

param_combinations	Recursively yield all combinations of param values

CHAPTER

THIRTEEN

EXPRESSION TRANSFORMATIONS (CPMPY.TRANSFORMATIONS)

Methods to transform CPMpy expressions in simpler CPMpy expressions

Input and output are always CPMpy expressions, so transformations can be chained and called multiple times, as needed.

A transformation can not modify expressions in place but in that case should create and return new expression objects. In this way, the expressions prior to the transformation remain intact, and could be used for other purposes too.

13.1 List of submodules

flatten_model	Flattening a model (or individual constraints) into 'flat normal form'.
get_variables	Returns an list of all variables in the model or expressions

CHAPTER

FOURTEEN

FAQ

Problem: I get the following error:

```
"IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`) and␣
↪integer or boolean arrays are valid indices"
```

Solution: Indexing an array with a variable is not allowed by standard numpy arrays, but it is allowed by cpmpy-numpy arrays. First convert your numpy array to a cpmpy-numpy array with the *cpm_array()* wrapper:

```
# x is a variable
X = intvar(0,3)

# Transforming a given numpy-array **m** into a CPMpy array
m = cpm_array(m)

# apply constraint
m[X] == 8
```

CHAPTER

FIFTEEN

LICENSE

This library is delivered under the MIT License, (see [LICENSE](https://github.com/tias/cppy/blob/master/LICENSE)).

PYTHON MODULE INDEX

C

INDEX