CPMpy

Release 0.5

Tias Guns

Nov 19, 2023

USAGE

1 Modeling and solving with CPMpy 3

2 Supported solvers 19

3 Open Source 33

Python Module Index 37

Index 39

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 is a Constraint Programming and Modeling library in Python, based on numpy, with direct solver access. Key features are:

- Easy to integrate with machine learning and visualisation libraries, because decision variables are numpy arrays.
- Solver-independent: transparently translating to CP, MIP, SMT and SAT solvers
- Incremental solving and direct access to the underlying solvers
- and much more…

CHAPTER

ONE

MODELING AND SOLVING WITH CPMPY

This page explains and demonstrates how to use CPMpy to model and solve combinatorial problems, so you can use it to solve for example routing, scheduling, assignment and other problems.

1.1 Installation

Installation is available through the `pip` python package manager. This will also install and use `ortools` as default solver:

```
pip install cpmpy
```

See installation instructions for more details.

1.2 Using the library

To conveniently use CPMpy in your python project, include it as follows:

```
from cpmpy import *
```

This will overload the built-in `any()`, `all()`, `min()`, `max()`, `sum()` functions, such that they create CPMpy expressions when used on decision variables (see below). This convenience comes at the cost of some overhead for all uses of these functions in your code.

You can also import it as a package, which does not overload the python built-ins:

```
import cpmpy as cp
```

We will use the latter in this document.

1.3 Decision variables

Constraint modeling consists of expressing *constraints* on *decision variables*, after which a solver will find a satisfying *assignment* to these decision variables.

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

```
import cpmpy as cp

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

Decision variables have a **domain**, a set of allowed values. For Boolean variables this is implicitly the values 'False' and 'True'. For Integer decision variables, you have to specify the lower-bound and upper-bound (`1` and `10` respectively above).

Decision variables have a **unique name**. You can set it yourself, otherwise a unique name will automatically be assigned to it. If you print `print(b, x)` decision variables, it will print the name. Did we already say the name must be unique? Many solvers use the name as unique identifier, and it is near-impossible to debug with non-uniquely named variables.

A solver will set the **value** of the decision variables for which it solved, if it can find a solution. You can retrieve it with `v.value()`. Variables are not tied to a solver, so you can use the same variable in multiple models and solvers. When a solve call finishes, it will overwrite the value of all its decision variables.

Finally, by providing a **shape** you automatically create a **numpy n-dimensional array** of decision variables. They automatically get their index appended to their name to ensure it is unique:

```
import cpmpy as cp

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

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

You can also call `v.value()` on these n-dimensional arrays, which will return an n-dimensional **numpy** array of values. And you can do vectorized operations and comparisons, like in regular numpy. As we will see below, this is very convenient and avoids having to write out many loops. It also makes it compatible with many existing scientific python tools, including machine learning and visualisation libraries, so a lot less glue code to write.

See the API documentation on variables for more detailed information.

1.4 Creating a model

A **model** is a collection of constraints over decision variables, optionally with an objective function. It represents a problem for which a solution must be found, e.g. by a solver. A solution is an assignment to the decision variables, such that each of the constraints is satisfied.

In CPMpy, the `Model()` object is a simple container that stores a list of CPMpy expressions representing constraints. It can also store a CPMpy expression representing an objective function that must be minimized or maximized. Constraints are added in the constructor, or using the built-in += addition operator that corresponds to calling the `__add__()` function.

Here is an example, where we explain how to express constraints in the next section:

```
import cpmpy as cp

# Decision variables
(x,y,z) = cp.intvar(1,10, shape=3)  # Python unpacks the array into the individual␣
↪variables
```

(continues on next page)

(continued from previous page)

```
# Initialise the model, here with 2 constraints
m = cp.Model(
    x == 1,
    x + y > 5
)

# Adding more constraints
m += (y - z != x)
m += (x + y + z <= 15)
# you can also add a list of constraints, which is interpreted as a conjunction of␣
↪constraints
m += [v <= 9 for v in [x,y,z]]

print(f"The model contains {len(m.constraints)} constraints")
print(m)  # pretty printing of the model, very useful for debugging
```

The `Model()` object has a number of other helpful functions, such as `to_file()` to store the model and `copy()` for creating a copy.

1.5 Expressing constraints

A constraint is a relation between decision variables that restricts what values these variables can take.

We now review the different types of constraints in CPMpy.

1.5.1 Logical constraints

To express **conjunction, disjunction and negation** of a constraint, we overwite the Python bitwise operators: `&` for conjunction (read as 'and'), `|` for disjunction (read as 'or') and `~` for negation (read as 'not').

Some examples:

```
import cpmpy as cp

# Decision variables
(a,b,c) = cp.boolvar(shape=3)

m = cp.Model(
    a | b,
    ~(a & c),
    (b | c) & ~a
)
```

Unfortunately, we can not overwite the `and`, `or` and `not` expression that we typically use in `if` expressions, so remember to use `&,|,~` instead. Also unfortunate is that Python bitwise operators have precedence over all other operators, so `a == 0 | b == 1` is **wrongly** interpreted by Python as `a == (0 | b) == 1` instead of the `(a == 0) | (b == 1)` that you probably intend. So make sure to **always write explicit brackets** when using `&,|,~`!

For **n-ary** conjunctions and disjunctions we overloaded the `all()` and `any()` functions:

```
import cpmpy as cp

# Decision variables
bv = cp.boolvar(shape=3)

m = cp.Model(
    cp.any([bv[0], bv[1], bv[2]]),
    cp.any(v for v in bv),  # equivalent to above
    cp.any(bv),  # equivalent to above
    ~cp.all(bv)
)
```

These functions accept manually created arrays, iterators or n-dimensional arrays alike.

For **equivalence**, also called reification, we overload the == comparison:

```
import cpmpy as cp

# Decision variables
a,b,c = cp.boolvar(shape=3)

m = cp.Model(
    a == b,  # equivalence: (a -> b) & (b -> a)
    a != b   # same as ~(a==b) and same as (a == ~b)
)
```

Finally for **implication** we decided that it would be most readable to introduce a function `implies()` to our (Boolean) expression objects, e.g.:

```
import cpmpy as cp

# Decision variables
a,b,c = cp.boolvar(shape=3)

m = cp.Model(
    a.implies(b),
    b.implies(a),
    a.implies(~c),
    (~c).implies(a)
)
```

For reverse implication, you switch the arguments yourself; it is difficult to read reverse implications out loud anyway. And as before, always use brackets around subexpressions to avoid surprises!

1.5.2 Simple comparison constraints

We overload Pythons comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. Comparisons are allowed between any CPMpy expressions as well as Boolean and integer constants.

On a technical note, we treat Booleans as a subclass of integer expressions. This means that a Boolean (output) expression can be used anywhere a numeric expression can be used, where `True` is treated as `1` and `False` as `0`. But the inverse is not true: integers can NOT be used with Boolean operators, even when you intialise their domain to (0,1) they are still not Boolean:

```
import cpmpy as cp

bv = cp.boolvar()
iv = cp.intvar(0,10)
iv01 = cp.intvar(0,1)

m = cp.Model(
    bv == True,          # allowed
    bv > 0,              # allowed but silly
    iv > 3,              # allowed
    iv != 6,             # allowed
    iv == True,          # allowed but avoid, means `iv == 1`
    iv == bv,            # allowed but avoid, means `(iv == 1) == bv`
    # bv & iv,           # not allowed, choose one of:
    bv & (iv == 1),      # allowed
    bv & (iv != 0),      # allowed
    # bv & iv01,         # not allowed, still an integer
)
```

CPMpy's array of decision variables is numpy-compatible, so it accepts **vectorized** operations on arrays of expressions:

```
import cpmpy as cp

iv = cp.intvar(0, 10, shape=3)

m = cp.Model(
    iv == 1,  # a vectorized operation, equivalent to:
    [iv[0] == 1, iv[1] == 1, iv[2] == 1]
)
```

You can convert a pure Python list of expressions into a numpy-compatible array by using `cpm_array()`:

```
import cpmpy as cp

x,y,z = cp.intvar(0, 10, shape=3)

m = cp.Model(
    # [x,y,z] == 1,  # does not work on plain Python arrays
    cp.cpm_array([x,y,z]) == 1  # does work, vectorized
)
```

1.5.3 Arithmetic constraints

We overload Python's built-in arithmetic operators `+,-,*,//,%`. These can be used to built arbitrarily nested numeric expressions, which can then be turned into a constraint by adding a comparison to it.

We also overwrite the built-in functions `abs(),sum(),min(),max()` which can be used to created numeric expressions. Some examples:

```
import cpmpy as cp

xs = cp.intvar(0, 10, shape=3, name="xs")
ys = cp.intvar(1, 10, shape=3, name="ys")

m = cp.Model(
    xs[0] - ys[0] == 5,
    cp.sum(xs) != 1,
    3*xs[0] < cp.abs(5 - cp.max(xs) + cp.min(ys))
)
```

All these operations can also be performed **vectorized** on arrays of the same shape, like in typical numpy code:

```
import cpmpy as cp
import numpy as np

xs = cp.intvar(0, 10, shape=3, name="xs")
w = np.array([1,3,-5])

m = cp.Model(
    cp.sum(w*xs) > 3,  # 1*xs[0] + 3*xs[1] + (-5)*xs[2] > 3
    xs + w != 0,  # [xs[0] + 1 != 0, xs[1] + 3 != 0, xs[2] + (-5) != 0]
    cp.max(xs - w) == np.arange(3),  # max(xs[0] - 1) == 0, max(xs[1] - 3) == 1,␣
↪max(xs[2] + 5) == 2]
)
```

Note that because of our overloading of `+,-,*,//` some numpy functions like `np.sum(some_array)` will also create a CPMpy expression when used on CPMpy decision variables. However, this is not guaranteed, and other functions like `np.max(some_array)` will not. To **avoid surprises**, you should hence always take care to call the CPMpy functions `cp.sum()`,cp.max()`etc. We did overload`some_cpm_array.sum()and.min()/.max()` (including the axis= argument), so these are safe to use.

1.5.4 Global constraints

You may wonder if you are allowed to use functions like `abs(),min(),max()` because some solvers might not have support for it? The answer is *yes you can use them*, because they are **global constraints**.

In constraint solving, a global constraint is a function that expresses a relation between decision variables. There are **two pathways when solving** a model with global constraints: 1) the solver natively supports them, or 2) the constraint modelling library automatically *decomposes* the constraint into an equivalent set of simpler constraints.

A good example is the `AllDifferent()` global constraint that ensures all its arguments have distinct values. `AllDifferent(x,y,z)` can be decomposed into `[x!=y, x!=z, y!=z]`. For AllDifferent, the decomposition consists of *n*(n-1)* pairwise inequalities, which are simpler constraints that most solvers support.

However, a solver that has specialised datastructures for this constraint specifically does not need to create the decomposition. Furthermore, for AllDifferent solvers can implement specialised algorithms that can propagate strictly stronger than the decomposed constraints can.

Global constraints

A non-exhaustive list of global constraints that are available in CPMpy is: `Xor()`, `AllDifferent()`, `AllDifferentExcept0()`, `Table()`, `Circuit()`, `Cumulative()`, `GlobalCardinalityCount()`.

For their meaning and more information on how to define your own global constraints, see the API documentation on global constraints. Global constraints can also be reified (e.g. used in an implication or equality constraint).

CPMpy will automatically decompose them if needed. If you want to see the decomposition yourself, you can call the `decompose()` function on them.

```
import cpmpy as cp
x = cp.intvar(1,4, shape=4, name="x")
b = cp.boolvar()
cp.Model(
    cp.AllDifferent(x),
    cp.AllDifferent(x).decompose(),  # equivalent: [(x[0]) != (x[1]), (x[0]) != (x[2]), .
↪..
    b.implies(cp.AllDifferent(x)),
    cp.Xor(b, cp.AllDifferent(x)),  # etc...
)
```

`decompose()` returns two arguments, one that represents the constraints and an optional one that defines any new variables needed. This is technical, but important to make negation work, if you want to know more check the the API documentation on global constraints.

Numeric global constraints

Coming back to the Python-builtin functions `min()`,`max()`,`abs()`, these are a bit special because they have a numeric return type. In fact, constraint solvers typically implement a global constraint `MinimumEq(args, var)` that represents `min(args) == var`, so it combines a numeric function with a comparison, where it will ensure that the bounds of the expressions on both sides satisfy the comparison relation.

However, CPMpy also wishes to support the expressions `min(xs) > v` as well as `v + min(xs) != 4` and other nested expressions.

In CPMpy we do this by instantiating min/max/abs as **numeric global constraints**. E.g. `min([x,y,z])` becomes `Minimum([x,y,z])` which inherits from `GlobalFunction` because it has a numeric return type. Our library will transform the constraint model, including arbitrarly nested expressions, such that the numeric global constraint is used in a comparison with a variable. Then, the solver will either support it, or we will call `decompose_comparison()` on the numeric global constraint, which will decompose e.g. `min(xs) == v`.

A non-exhaustive list of **numeric global constraints** that are available in CPMpy is: `Minimum()`, `Maximum()`, `Count()`, `Element()`.

For their meaning and more information on how to define your own global constraints, see the API documentation on global functions.

```
import cpmpy as cp
x = cp.intvar(1,4, shape=4, name="x")
s = cp.SolverLookup.get("ortools")
print(s.transform(cp.min(x) + cp.max(x) - 5 > 2*cp.Count(x, 2)))
# [(sum([IV5, IV6, -5])) > (IV4),
#  (min(x[0],x[1],x[2],x[3])) == (IV5), (max(x[0],x[1],x[2],x[3])) == (IV6),
#  (sum([2] * [IV3])) == (IV4),
#  (sum([BV0, BV1, BV2, BV3])) == (IV3),
```

(continues on next page)

(continued from previous page)

```
#  (~BV0) -> (x[0] != 2), (BV0) -> (x[0] == 2),
#  (~BV1) -> (x[1] != 2), (BV1) -> (x[1] == 2),
#  (~BV2) -> (x[2] != 2), (BV2) -> (x[2] == 2),
#  (~BV3) -> (x[3] != 2), (BV3) -> (x[3] == 2)]
```

The Element numeric global constraint

The `Element(Arr,Idx)` global function enforces that the result equals `Arr[Idx]` with `Arr` an array of constants or variables (the first argument) and `Idx` an integer decision variable, representing the index into the array.

```
import cpmpy as cp

arr = cp.intvar(1,10, shape=4)
idx = cp.intvar(0,len(arr)-1)  # indexing is offset 0

m = cp.Model(
    cp.AllDifferent(arr),
    arr[idx] == 2
)
m.solve()
print(f"arr: {arr.value()}, idx: {idx.value()}, val: {arr[idx].value()}")
# example output -- arr: [2 1 3 4], idx: 0, val: 2
```

The `arr[idx]` works because `arr` is a CPMpy `NDVarArray()` and we overloaded the `__getitem__()` python function. It even supports multi-dimensional access, e.g. `arr[idx1,idx2]`.

This does not work on NumPy arrays though, as they don't know CPMpy. So you have to **wrap the array** in our `cpm_array()` or call `Element()` directly:

```
import numpy as np
import cpmpy as cp

arr = np.arange(4)  # array([0, 1, 2, 3])
idx = cp.intvar(0,len(arr))  # indexing is offset 0

m = cp.Model()
#m += (arr[idx] == 2)             # does not work, numpy does not know what to do
# IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`) and␣
↪integer or boolean arrays are valid indices

cparr = cp.cpm_array(arr)         # wrap in CPMpy array
m += (cparr[idx] == 2)            # works

m += (cp.Element(arr, idx) == 2)  # also works, identical to above

m.solve()
print(f"arr: {arr.value()}, idx: {idx.value()}, val: {arr[idx].value()}")
# arr: [0 1 2 3], idx: 2, val: 2
```

1.6 Objective functions

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.

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

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

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

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

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

1.7 Solving a model

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

The return value of `solve()` is a Boolean indicating whether a solution was found. So regardless of whether it was a satisfaction or optimisation problem or with a timeout, it returns true if 'a' solution has been found in the process.

To know the exact solver state and runtime after solve, call `status()`. In case of an optimisation problem, you can get the objective value of the solution with `objective_value()`.

```
import cpmpy as cp
xs = cp.intvar(1,10, shape=3)
m = cp.Model(cp.AllDifferent(xs), maximize=cp.sum(xs))

hassol = m.solve()
print("Status:", m.status())  # Status: ExitStatus.OPTIMAL (0.03033301 seconds)
if hassol:
    print(m.objective_value(), xs.value())  # 27 [10  9  8]
else:
    print("No solution found.")
```

1.8 Finding all solutions

You can also conveniently use CPMpy to find all solutions using the `solveAll()` function:

```
import cpmpy as cp
x = cp.intvar(0, 3, shape=2)
m = cp.Model(x[0] > x[1])

n = m.solveAll()
print("Nr of solutions:", n)  # Nr of solutions: 6
```

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

It has a `display=...` argument that can be used to display expressions or as a callback, as well as the `solution_limit=...` argument to set a solution limit. It also accepts any named argument, like `time_limit=...`, that the underlying solver accepts.

```
n = m.solveAll(display=[x,cp.sum(x)], solution_limit=3)
# [array([1, 0]), 1]
# [array([2, 0]), 2]
# [array([3, 0]), 3]
```

There is much more to say on enumerating solutions and the use of callbacks or blocking clauses. See the the detailed documentation on finding multiple solutions.

1.9 Debugging a model

If the solver is complaining about your model, then a good place to start debugging is to **print** the model you have created, or the individual constraints. If they look fine (e.g. no integers, or shorter or longer expressions then what you intended) and you don't know which constraint specifically is causing the error, then you can feed the constraints incrementally to the solver you are using:

```
import cpmpy as cp

cons = []  # ... imagine a list of constraints
print(cons)

m = cp.Model(cons)  # any model created
# visually inspect that the constraints match what you wanted to express
# e.g. if you wrote `all(x)` instead of `cp.all(x)` it will contain 'True' instead of the␣
↪conjunction
print(m)

s = cp.SolverLookup.get("ortools")
# feed the constraints one-by-one
for c in m.constraints:
    s += c  # add the constraints incrementally until you hit the error
```

If that is not sufficient or you want to debug an unexpected (non)solution, have a look at our detailed Debugging guide.

1.10 Selecting a solver

The default solver is OR-Tools CP-SAT, an award winning constraint solver. But CPMpy supports multiple other solvers: a MIP solver (gurobi), SAT solvers (those in PySAT), the Z3 SMT solver, even a knowledge compiler (PySDD) and any CP solver supported by the text-based MiniZinc language.

See the full list of solvers known by CPMpy with:

```
import cpmpy as cp
cp.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 specify a solvername when calling `solve()` on a model:

```
import cpmpy as cp
x = cp.intvar(0,10, shape=3)
m = cp.Model(cp.sum(x) <= 5)
# use named solver
m.solve(solver="minizinc:chuffed")
```

Note that for solvers other than "ortools", you will need to **install additional package(s)**. You can check if a solver, e.g. "minizinc", is supported by calling `cp.SolverLookup.get("gurobi")` and it will raise a helpful error if it is not yet installed on your system. See the API documentation of the solver for detailed installation instructions.

1.11 Model versus solver interface

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. So `m.solve("ortools")` is equivalent to:

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

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.

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

On a technical note, remark that a solver object does not modify the Model object with which it is initialised. So adding constraints to the solver does not add them to that model, and calling `s.solve()` does not update the status of `m.status()`, only of `s.status()`.

1.12 Setting solver parameters

Now lets use our solver-specific powers. For example, with `m` a CPMpy Model(), you can do the following to make or-tools use 8 parallel cores and print search progress:

```
import cpmpy as cp
s = cp.SolverLookup.get("ortools", m)
# we are operating on the ortools interface here
s.solve(num_search_workers=8, log_search_progress=True)
```

Modern CP-solvers support a variety of hyperparameters. (See the full list of OR-tools parameters for example). Using the solver interface, any parameter that the solver supports can be passed using the `.solve()` call. These parameters will be posted to the native solver 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.

1.13 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. That is why the debugging trick of posting it one-by-one works.

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
# the underlying gurobi instance is reused, only the new constraint is added to it.
# gurobi is an incremental solver and will look for solutions starting from the previous␣
↪one.
gs.solve()
```

Technical note: OR-Tools its model representation is incremental but its solving itself is not (yet?). Gurobi and the PySAT solvers are fully incremental, as is Z3. The text-based MiniZinc language is not incremental.

1.14 Using solver-specific CPMpy features

We sometimes add solver-specific functions to the CPMpy interface, for convenient access. Two examples of this are `solution_hint()` and `get_core()` which is supported by the OR-Tools and PySAT solvers and interfaces. Other solvers may work differently and 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:

```
import cpmpy as cp
x = cp.intvar(0,10, shape=3)
s = cp.SolverLookup.get("ortools")
s += cp.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.

1.15 Direct solver access

Some solvers implement more constraints then available in CPMpy. But CPMpy offers direct access to the underlying solver, so there are two ways to post such solver-specific constraints.

1.15.1 DirectConstraint

The `DirectConstraint` will directly call a function of the underlying solver, when the constraint is added to a CPMpy solver.

You provide the DirectConstraint with the name of the function you want to call, as well as the arguments:

```
import cpmpy as cp
iv = cp.intvar(1,9, shape=3)

s =  cp.SolverLookup.get("ortools")
s += cp.AllDifferent(iv)
s += cp.DirectConstraint("AddAllDifferent", iv)  # a DirectConstraint equivalent to the␣
↪above for OR-Tools
```

This requires knowledge of the API of the underlying solver, as any function name that you give to it will be called. The only special thing that the DirectConstraint does, is automatically translate any CPMpy variable in the arguments to the native solver variable.

Note that any argument given will be checked for whether it needs to be mapped to a native solver variable. This may give errors on complex arguments, or be inefficient. You can tell the `DirectConstraint` not to scan for variables with the `novar` argument, for example:

```
import cpmpy as cp
trans_vars = cp.boolvar(shape=4, name="trans")

s = cp.SolverLookup.get("ortools")

trans_tabl = [ # corresponds to regex 0* 1+ 0+
    (0, 0, 0),
    (0, 1, 1),
    (1, 1, 1),
    (1, 0, 2),
    (2, 0, 2)
]
s += cp.DirectConstraint("AddAutomaton", (trans_vars, 0, [2], trans_tabl),
                         novar=[1, 2, 3])  # optional, what arguments not to scan for␣
↪vars
```

A minimal example of the DirectConstraint for every supported solver is in the test suite.

The `DirectConstraint` is a very powerful primitive to get the most out of specific solvers. See the following examples: nonogram_ortools.ipynb which uses a helper function that generates automatons with DirectConstraints; vrp_ortools.py demonstrating ortools' newly introduced multi-circuit global constraint through DirectConstraint; and pctsp_ortools.py that uses a DirectConstraint to use OR-Tools circuit to post a sub-circuit constraint as needed for this price-collecting TSP variant.

1.15.2 Directly accessing the underlying solver

The `DirectConstraint("AddAllDifferent", iv)` is equivalent to the following code, which demonstrates that you can mix the use of CPMpy with calling the underlying solver directly:

```
import cpmpy as cp

iv = cp.intvar(1,9, shape=3)

s = cp.SolverLookup.get("ortools")

s += AllDifferent(iv)  # the traditional way, equivalent to:
s.ort_model.AddAllDifferent(s.solver_vars(iv))  # directly calling the API, has to be␣
↪with native variables
```

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 DirectConstraint, or any CPMpy constraint.

While directly calling the solver offers a lot of freedom, it is a bit more cumbersome as you have to map the variables manually each time. Also, you no longer have a declarative model that you can pass along, print or inspect. In contrast, a `DirectConstraint` is a CPMpy expression so it can be part of a model like any other CPMpy constraint. Note that it can only be used as top-level (non-nested, non-reified) constraint.

1.16 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.

1.16.1 Built-in tuners

The tools directory contains a utility to efficiently search through the hyperparameter space defined by the solvers `tunable_params`.

Solver interfaces not providing the set of tunable parameters can still be tuned by using this utility and providing the parameter (values) yourself.

```
import cpmpy as cp
from cpmpy.tools import ParameterTuner

model = cp.Model(...)

tunables = {
    "search_branching":[0,1,2],
    "linearization_level":[0,1],
    'symmetry_level': [0,1,2]}
defaults = {
    "search_branching": 0,
    "linearization_level": 1,
    'symmetry_level': 2
}

tuner = ParameterTuner("ortools", model, tunables, defaults)
best_params = tuner.tune(max_tries=100)
best_runtime = tuner.best_runtime
```

This utlity is based on the SMBO framework and speeds up the search by starting from the default configuration, and implementing adaptive capping meaning that the best runtime is used as timeout to avoid wasting time.

The parameter tuner is based on the following publication:

> Ignace Bleukx, Senne Berden, Lize Coenen, Nicholas Decleyre, Tias Guns (2022). Model-Based Algorithm Configuration with Adaptive Capping and Prior Distributions. In: Schaus, P. (eds) Integration of Constraint Programming, Artificial Intelligence, and Operations Research. CPAIOR 2022. Lecture Notes in Computer Science, vol 13292. Springer, Cham. https://doi.org/10.1007/978-3-031-08011-1_6

Another built-in tuner is `GridSearchTuner`, which does random gridsearch (with adaptive capping).

1.16.2 External tuners

You can also use external hyperparameter optimisation libraries, such as `hyperopt`:

```
from hyperopt import tpe, hp, fmin
import cpmpy as cp

# model = Model(...)
```

(continues on next page)

(continued from previous page)

```
def time_solver(model, solver, param_dict):
    s = cp.SolverLookup.get(solver, model)
    s.solve(**param_dict)
    return s.status().runtime

space = {
    'cp_model_probing_level': hp.choice('cp_model_probing_level', [0, 1, 2, 3]),
    'linearization_level': hp.choice('linearization_level', [0, 1, 2]),
    'symmetry_level': hp.choice('symmetry_level', [0, 1, 2]),
    'search_branching': hp.choice('search_branching', [0, 1, 2]),
}

best = fmin(
    fn=lambda p: time_solver(model, "ortools", p), # Objective Function to optimize
    space=space, # Hyperparameter's Search Space
    algo=tpe.suggest, # Optimization algorithm (representative TPE)
    max_evals=10 # Number of optimization attempts
)
print(best)
time_solver(model, "ortools", best)
```

CHAPTER

TWO

SUPPORTED SOLVERS

CPMpy can translate to many different solvers, and even provides direct access to them.

To make clear how well supported and tested these solvers are, we work with a tiered classification:

- **Tier 1 solvers: passes all internal tests, passes our bigtest suit, will be fuzztested in the near future**
  - "ortools" the OR-Tools CP-SAT solver
  - "pysat" the PySAT library and its many SAT solvers ("pysat:glucose4", "pysat:lingeling", etc)
- **Tier 2 solvers: passes all internal tests, might fail on edge cases in bigtest**
  - "minizinc" the MiniZinc modeling system and its many solvers ("minizinc:gecode", "minizinc:chuffed", etc)
  - "z3" the SMT solver and theorem prover
  - "gurobi" the MIP solver
  - "PySDD" a Boolean knowledge compiler
  - "exact" the Exact integer linear programming solver
- **Tier 3 solvers: they are work in progress and live in a pull request**
  - "gcs" the Glasgow Constraint Solver

We hope to upgrade many of these solvers to higher tiers, as well as adding new ones. Reach out on github if you want to help out.

2.1 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.

2.1.1 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).

2.1.2 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).

2.1.3 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 ;)

2.1.4 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 not what you intended to express, as described in 'Debugging a modeling error'.

If that does not help, try printing the 'transformed' **constraints**, the way that the solver actually sees them, including decompositions and rewrites:

```
s = SolverLookup.get("ortools")  # or whatever solver you are using
print(s.transform(model.constraints))
```

Note that you can also print individual expressions like this, e.g. `print(s.transform(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:

```
s = SolverLookup.get("ortools")  # or whatever solver you are using
ct = s.transform(model.constraints)
from cpmpy.transformations.get_variables import print_variables
print_variables(ct)
print(ct)
```

Printing the **objective** as the solver sees it requires you to look into the solver interface code of that solver. However, the following is a good first check that can already reveal potentially problematic things:

```
s = SolverLookup.get("ortools")  # or whatever solver you are using
from cpmpy.transformations.flatten_model import flatten_objective
(obj_var, obj_expr) = flatten_objective(model.objective)
print(f"Optimizing {obj_var} subject to", s.transform(obj_expr))
```

Automatically minimising the UNSAT program

If the above is unwieldy because your constraint problem is too large, then consider automatically reducing it to a 'Minimal Unsatisfiable Subset' (MUS).

This is now part of our standard tools, that you can use as follows:

```
from cpmpy.tools import mus

x = boolvar(shape=3, name="x")
model = Model(
    x[0],
    x[0] | x[1],
    x[2].implies(x[1]),
```

(continues on next page)

(continued from previous page)

```
    ~x[0],
    )

unsat_cons = mus(model.constraints)
```

With this smaller set of constraints, repeat the visual inspection steps above.

(Note that for an UNSAT problem there can be many MUSes, the `examples/advanced/` folder has the MARCO algorithm that can enumerate all MSS/MUSes.)

2.1.5 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 above on 'Automatically minimising the UNSAT program' to better understand it.

2.1.6 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, check out our advanced example on stepwise OCUS explanations

2.2 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.

2.2.1 `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())}")
```

See the set_game.ipynb for an example of how we use it as a callback to call a plotting function, to plot all the solutions as they are found.

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.")
```

2.2.2 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

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)
```

2.2.3 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():
```

(continues on next page)

(continued from previous page)

```
    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]))
```

2.2.4 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())
```

2.3 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 (MUS) and QuickXplain-like tools to help debugging.

In our tools we implemented a simple MUS deletion based algorithm, using assumption variables.

```
from cpmpy.tools import mus

print(mus(m.constraints))
```

We welcome any additional examples that use CPMpy in this way!! Here is one example: the MARCO algorithm for enumerating all MUS/MSSes. Here is another: a stepwise explanation algorithm for SAT problems (implicit hitting-set based)

One OR-TOOLS specific caveat is that this particular (default) solver its 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.

2.4 Developer guide

CPMpy is an open source project and we are happy for you to read and change the code as you see fit.

This page introduces how to get started on developing on CPMpy itself, with a focus on sharing these changes back with us for inclusion.

2.4.1 Setting up your development environment

The easiest is to use the pip to do an 'editable install' of your local CPMpy folder.

```
pip install --editable .
```

With that, any change you do there (including checking out different branches) is automatically used wherever you use CPMpy on your system.

2.4.2 Running the test suite

We only accept pull requests that pass all the tests. In general, you want to know if your changes screwed up another part. From your local CPMpy folder, execute:

```
python -m pytest tests/
```

This will run all tests in our `tests/` folder.

You can also run an individual test, as such (e.g. when wanting to test a new solver):

```
python -m pytest tests/test_solvers.py
```

2.4.3 Code structure

- *tests/* contains the tests
- *docs/* contains the docs. Any change there is automatically updated, with some delay, on https://cpmpy.readthedocs.io/
- *examples/* our examples, always happy to include more
- *cpmpy/* the python module that you install when doing `pip install cpmpy`

The module is structured as such:

- *model.py* contains the omnipresent `Model()` container
- *expressions/* Classes and functions that represent and create expressions (constraints and objectives)
- *solvers/* CPMpy interfaces to (the Python API interface of) solvers
- *transformations/* Methods to transform CPMpy expressions in other CPMpy expressions
- *tools/* Set of independent tools that users might appreciate.

The typical flow in which these submodules are used when using CPMpy is: the user creates *expressions* which they put into a *model* object. This is then given to a *solver* object to solve, which will first *transform* the expressions into expressions that it supports, which it then posts to the Python API interface of that particular solver.

Tools are not part of the core of CPMpy. They are additional tools that *use* CPMpy, e.g. for debugging, parameter tuning etc.

2.4.4 Github practices

When filing a bug, please add a small case that allows us to reproduce it. If the testcase is confidential, mail Tias directly.

Only documentation changes can be directly applied on the master branch. All other changes should be submitted as a pull request.

When submitting a pull request, make sure it passes all tests.

When fixing a bug, you should also add a test that checks we don't break it again in the future (typically, the case from the bugreport).

We are happy to do code reviews and discuss good ways to fix something or add a new feature. So do not hesitate to create a pull request for work-in-progress code. In fact, almost all pull requests go through at least 1 revision iteration.

2.5 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 to 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
- `transform()` where you call the necessary transformations in `cpmpy.transformations` to transform CPMpy expressions to those that the solver supports
- `__add__()` where you call transform and map the resulting CPMpy expressions that the solver supports, to API function calls on the underlying solver
- `solveAll()` optionally, if the solver natively supports solution enumeration

2.5.1 Transformations and posting constraints

CPMpy solver interfaces are *eager*, meaning that any CPMpy expression given to it (through `__add__()`) is immediately transformed (throught `transform()`) and then posted to the solver.

CPMpy is designed to separate *transforming* arbitrary CPMpy expressions to constraints the solver supports, from actually *posting* the supported constraints directly to the solver.

For example, a SAT solver only accepts clauses (disjunctions) over Boolean variables as constraints. So, its `transform()` method has the challenge 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 solver-independent **transformation functions** in the `cpmpy/transformations/` directory that can achieve that and more. You hence only need to chain the right transformations in the solver's `transform()` method. It is best to look at a solver accepting a similar input, to see what transformations (and in what order) that one uses.

The `__add__()` method will first call this `transform()`. This will return a list of CPMpy 'or' expression over decision variables. It then only has to iterate over those and call the solver its native API to create such clauses. All other constraints may not be directly supported by the solver, and can hence be rejected.

So for any solver you wish to add, chances are that most of the transformations you need are already implemented. Any solver can use any transformation in any order that the transformations allow. If you need additional transformations, or want to know how they work, read on.

2.5.2 Stateless transformation functions

Because CPMpy solver interfaces transform and post constraints *eagerly*, they can be used *incremental*, meaning that you can add some constraints, call `solve()` add 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 to. 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/`.

2.5.3 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 fictional pseudo-code of such an API, which is heavily inspired on the OR-Tools 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)
```

(continues on next page)

(continued from previous page)

```
    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);
    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!

2.5.4 Testing your solver

The CPMpy package provides a large testsuite on which newly added solvers can be tested. Note that for this testsuite to work, you need to add your solver to the `SolverLookup` utility. This is done by adding an import statement in `/solvers/__init__.py` and adding an entry in the list of solvers in `/solvers/utils.py`.

To run the testsuite on your solver, go to `/tests/test_constraints.py` and set `SOLVERNAMES` to the name of your solver. By running the file, every constraint allowed by the Flat Normal Form will be generated and posted to your solver interface. As not every solver should support all possible constraints, you can exclude some using the `EXCLUDE_GLOBAL`, `EXCLUDE_OPERATORS` and `EXCLUDE_IMPL` dictionaries. After posting the constraint, the answer of your solver is checked so you will both be able to monitor when your interface crashes or when a translation to the solver is incorrect.

2.5.5 Tunable hyperparameters

CPMpy offers a tool for searching the best hyperparameter configuration for a given model on a solver (see corresponding documentation). Solver wanting to support this tool should add the following attributes to their solver interface: `tunable_params` and `default_params` (see ortools for an example).

CHAPTER

THREE

OPEN SOURCE

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

CPMpy has the open-source Apache 2.0 license and is run as an open-source project. All discussions happen on Github, even between direct colleagues, and all changes are reviewed through pull requests.

Join us! We welcome any feedback and contributions. You are also free to reuse any parts in your own project. A good starting point to contribute is to add your models to the examples folder.

Are you a solver developer? We are keen to integrate solvers that have a python API on pip. If this is the case for you, or if you want to discuss what it best looks like, do contact us!

3.1 Expressions (`cpmpy.expressions`)

Classes and functions that represent and create expressions (constraints and objectives)

3.1.1 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.
`globalfunctions`	Using global functions
`python_builtins`	Overwrites a number of python built-ins, so that they work over variables as expected.
`utils`	Internal utilities for expression handling.

3.2 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()

3.2.1 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.

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

3.3 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

3.3.1 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
`z3`	Interface to z3's API
`exact`	Interface to Exact
`utils`	Utilities for handling solvers

3.3.2 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
CPM_z3	Interface to z3's API
CPM_exact	Interface to the Python interface of Exact

3.3.3 List of functions

param_combinations	Recursively yield all combinations of param values

3.4 Expression transformations (cpmpy.transformations)

Methods to transform CPMpy expressions in other 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.

3.4.1 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

PYTHON MODULE INDEX

C

cpmpy.expressions, 33
cpmpy.model, 33
cpmpy.solvers, 35
cpmpy.transformations, 36

INDEX

C

copy() (*cpmpy.model.Model method*), 34
cpmpy.expressions
    module, 33
cpmpy.model
    module, 33
cpmpy.solvers
    module, 35
cpmpy.transformations
    module, 36

F

from_file() (*cpmpy.model.Model static method*), 34

M

maximize() (*cpmpy.model.Model method*), 34
minimize() (*cpmpy.model.Model method*), 34
Model (*class in cpmpy.model*), 34
module
    cpmpy.expressions, 33
    cpmpy.model, 33
    cpmpy.solvers, 35
    cpmpy.transformations, 36

O

objective() (*cpmpy.model.Model method*), 34
objective_value() (*cpmpy.model.Model method*), 34

S

solve() (*cpmpy.model.Model method*), 34
solveAll() (*cpmpy.model.Model method*), 35
status() (*cpmpy.model.Model method*), 35

T

to_file() (*cpmpy.model.Model method*), 35