pychoco

Release 0.1.1

Dimitri Justeau-Allaire, Charles Prud'homme

Oct 06, 2022

CONTENTS:

Python bindings for the Choco Constraint programming solver (https://choco-solver.org/).

Choco-solver is an open-source Java library for Constraint Programming (see https://choco-solver.org/). It comes with many features such as various types of variables, various state-of-the-art constraint, various search strategies, etc.

The PyChoco library uses a *native-build* of the original Java Choco-solver library, in the form of a shared library, which means that it can be used without any JVM. This native-build is created with GraalVM (https://www.graalvm.org/) native-image tool.

We heavily relied on JGraphT Python bindings (https://python-jgrapht.readthedocs.io/) source code to understand how such a thing could be achieved, so many thanks to JGraphT authors!

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

We are still in the process of implementing and releasing PyChoco. So currently the only way to install it and try it is to follow the entire build-from-source process. However, we plan to release pre-built Python wheels for various operating systems. Stay tuned!

1.1.1 Installation from PyPI

We automatically build 64-bit wheels for Python versions 3.6, 3.7, 3.8, 3.9, and 3.10 on Linux, Windows and MacOSX. They can be directly downloaded from PyPI (https://pypi.org/project/pychoco/) or using pip:

```
$ pip install pychoco
```

1.1.2 Build from source

The following system dependencies are required to build PyChco from sources:

- GraalVM >= 20 (see https://www.graalvm.org/)
- Native Image component for GraalVM (see https://www.graalvm.org/22.1/reference-manual/native-image/)
- Apache Maven (see https://maven.apache.org/)
- Python >= 3.6 (see https://www.python.org/)
- SWIG >= 3 (see https://www.swig.org/)

Once these dependencies are satisfied, clone the current repository:

```
$ git clone --recurse-submodules https://github.com/dimitri-justeau/pychoco.git
```

The *–recurse-submodules* is necessary as the *choco-solver-capi* is a separate git project included as a submodule (see https://github.com/dimitri-justeau/choco-solver-capi). It contains all the necessary to compile Choco-solver as a shared native library using GraalVM native-image.

Ensure that the *$JAVA_HOME* environment variable is pointing to GraalVM, and from the cloned repository execute the following command:

```
$ sh build.sh
```

This command will compile Choco-solver into a shared native library and compile the Python bindings to this native API using SWIG.

Finally, run:

```
$ pip install .
```

And voilà !

1.2 Quickstart

Pychoco's API is quite close to Choco's Java API. The first thing to do is to import the library and create a model object:

```
from pychoco import Model

model = Model("My Choco Model")
```

Then, you can use this model object to create variables:

```
intvars = model.intvars(10, 0, 10)
sum_var = model.intvar(0, 100)
```

You can also create views from this Model object:

```
b6 = model.int_ge_view(intvars[6], 6)
```

Create and post (or reify) constraints:

```
model.all_different(intvars).post()
model.sum(intvars, "=", sum_var).post()
b7 = model.arithm(intvars[7], ">=", 7).reify()
```

Solve your problem:

```
model.get_solver().solve()
```

And retrieve the solution:

```
print("intvars = {}".format([i.get_value() for i in intvars]))
print("sum = {}".format(sum_var.get_value()))
print("intvar[6] >= 6 ? {}".format(b6.get_value()))
print("intvar[7] >= 7 ? {}".format(b7.get_value()))

> "intvars = [3, 5, 9, 6, 7, 2, 0, 1, 4, 8]"
> "sum = 45"
> "intvar[6] >= 6 ? False"
> "intvar[7] >= 7 ? False"
```

1.3 API

1.3.1 Model

The model is the core component of PyChoco. A model is created using the *Model()* constructor, and it is the entry point to create variables, constraints, and solve problems.

1.3.2 Variables

A variable is an unknown, mathematically speaking. The goal of a resolution is to assign a value to each variable. The domain of a variable –set of values it may take– must be defined in the model. Currently, PyChoco supports boolean variables (BoolVar), integer variables (IntVar), and set variables (SetVar). Variables are created using a *Model* object (see *Model*). When creating a variable, the user can specify a name to help reading the output.

Variable

The *Variable* class is the superclass of all classes, it contains generic methods and property that are common to all types of variables.

IntVar

Integer variables represent a integer value, and can be created from a Model object using the following methods:

Integer variables also include additional parameters and methods to the generic Variable class:

Operations between IntVars

We took advantage of operators overloading in Python to provide some shortcuts in pychoco, so you can use the following operators between IntVars and ints.

- *c = a + b*: c is and IntVar constrained to be equal to a + b (see *arithm* constraint in *Constraints*).
- *c = a - b*: c is and IntVar constrained to be equal to a - b (see *arithm* constraint in *Constraints*).
- *c = a * b*: c is and IntVar constrained to be equal to a * b (see *arithm* constraint in *Constraints*).
- *c = a / b*: c is and IntVar constrained to be equal to a / b (see *arithm* constraint in *Constraints*).
- *c = -a*: c is an *int_minus_view* (see ref:*views*)
- *c = a % b*: c is the result rest of the integer division betwen a and b (see *mod* constraint in *Constraints*).
- *c = a ** c* c is equal to pow(a, c), c must be an int (see *pow* constraint in *Constraints*).
- *c = a == b* c is a BoolVar, which is True only if a == b.
- *c = a <= b* c is a BoolVar, which is True only if a <= b.
- *c = a < b* c is a BoolVar, which is True only if a < b.
- *c = a >= b* c is a BoolVar, which is True only if a >= b.
- *c = a > b* c is a BoolVar, which is True only if a > b.
- *c = a != b* c is a BoolVar, which is True only if a != b.

BoolVar

Boolean variables represent a boolean value (0/1 or False/True). They are a special case of integer variables where the domain is restricted to [0, 1], and can be created from a Model object using the following methods:

Boolean variables also include additional parameters and methods to the generic Variable class:

Operations between BoolVars

We took advantage of operators overloading in Python to provide some shortcuts in pychoco, so you can use the following operators between BoolVars and bools.

- *b = b1 & b2*: b is a BoolVar which is True only if b1 and b2 are True (see *and_* constraint in *Constraints*).
- *b = b1 | b2*: b is a BoolVar which is True only if b1 or b2 is True (see *or_* constraint in *Constraints*).
- *b = ~b1*: b is a *bool_not_view* over b1 (see *Views*).
- *b = b1 == b2* is a BoolVar which is True only if *b1 == b2*.
- *b = b1 != b2* is a BoolVar which is True only if *b1 != b2*.

SetVar

Set variables represent a set of integers, which value must belong to a set interval [lb, ub]. The lower bound lb is the set of mandatory values (or kernel) for any instantiation of the variable, while the upper bound ub is the set of potential values (or envelope) for any instantiation of the variable. Set variables can be created from a model object using the following method:

Set variables also include additional parameters and methods the generic Variable class:

GraphVar

Graph variables represent a graph (directed or undirected), which value must belong to a graph interval [lb, ub]. The lower bound lb (or kernel) is a graph that must be included in any instantiation of the variable, while the upper bound ub (or envelope) is such that any instantiation of the variable is a subgraph of it.

The bounds of a graph variable must be created using the graph API of pychoco (see below).

Undirected Graph variables can be created from a model object using the following methods:

Undirected variables also include additional parameters and methods the generic Variable class:

Directed Graph variables can be created from a model object using the following methods:

Directed variables also include additional parameters and methods the generic Variable class:

UndirectedGraph API

The *create_undirected_graph* factory function allows to instantiate a directed graph from a list of nodes and a list of edges:

This function returns an *UndirectedGraph* object:

DirectedGraph API

The *create_directed_graph* factory function allows to instantiate a directed graph from a list of nodes and a list of edges:

This function returns a *DirectedGraph* object:

1.3.3 Constraints

A constraint is a logic formula defining allowed combinations of values for a set of variables (see *Variables*), i.e., restrictions over variables that must be respected in order to get a feasible solution. A constraint is equipped with a (set of) filtering algorithm(s), named propagator(s). A propagator removes, from the domains of the target variables, values that cannot correspond to a valid combination of values. A solution of a problem is a variable-value assignment verifying all the constraints.

Constraints are directly declared from a *Model* object (see *Model*).

Integer and boolean constraints

All constraints over integer and boolean variables are declared in the *IntConstraintFactory* abstract class, which is implemented by the *Model* class.

absolute

all_different

all_different_except_0

all_different_prec

all_equal

among

and

argmax

argmin

arithm

at_least_n_values

at_most_n_values

bin_packing

bits_int_channeling

bools_int_channeling

circuit

clauses_int_channeling

cost_regular

count

cumulative

decreasing

diff_n

distance

div

element

global_cardinality

increasing

int_value_precede_chain

inverse_channeling

keysort

knapsack

lex_chain_less

lex_chain_less_eq

lex_less

lex_less_eq

max

mddc

member

min

mod

multi_cost_regular

n_values

not

not_all_equal

not_member

or

path

pow

regular

scalar

sort

square

sub_circuit

sub_path

sum

table

times

tree

Set constraints

All constraints over set variables in the *SetConstraintFactory* abstract class, which is implemented by the *Model* class. Set constraints have the *set_* prefix, indeed, as several set constraints have the same name as int constraints, we made the choice to semantically distinguish them, contrarily to the Choco Java API, as method Python does not support method overloading.

set_all_different

set_all_disjoint

set_all_equal

set_bools_channeling

set_disjoint

set_element

set_intersection

set_ints_channeling

set_inverse_set

set_le

set_lt

set_max

set_max_indices

set_member_int

set_member_set

set_min

set_min_indices

set_nb_empty

set_not_empty

set_not_member_int

set_offset

set_partition

set_subset_eq

set_sum

set_sum_element

set_symmetric

set_union

set_union_indices

Graph constraints

All constraints over graph variables in the *GraphConstraintFactory* abstract class, which is implemented by the *Model* class. Graph constraints have the *graph_* prefix, indeed, as method Python does not support method overloading, we made the choice to semantically distinguish them to avoid method name conflicts.

graph_anti_symmetric

graph_biconnected

graph_connected

graph_cycle

graph_degrees

graph_diameter

graph_directed_forest

graph_directed_tree

graph_edge_channeling

graph_forest

graph_in_degrees

graph_loop_set

graph_max_degree

graph_max_degrees

graph_max_in_degree

graph_max_in_degrees

graph_max_out_degree

graph_max_out_degrees

graph_min_degree

graph_min_degrees

graph_min_in_degree

graph_min_in_degrees

graph_min_out_degree

graph_min_out_degrees

graph_nb_cliques

graph_nb_connected_components

graph_nb_edges

graph_nb_loops

graph_nb_nodes

graph_nb_strongly_connected_components

graph_neighbors_channeling

graph_no_circuit

graph_no_cycle

graph_node_channeling

graph_node_neighbors_channeling

graph_node_predecessors_channeling

graph_node_successors_channeling

graph_nodes_channeling

graph_out_degrees

graph_reachability

graph_size_connected_components

graph_size_max_connected_components

graph_size_min_connected_components

graph_strongly_connected

graph_subgraph

graph_successors_channeling

graph_symmetric

graph_transitivity

graph_tree

1.3.4 Views

The concept of views in Constraint Programming is halfway between variables and constraints. Specifically, a view is a special kind of variable that does not declare any domain, but instead relies on one or several other variables through a logical relation. From a modelling perspective, a view can be manipulated exactly as any other variable. In pychoco, the only difference that you will notice is that the *is_view()* method will return True when a variable is actually a view.

Views are directly declared from a *Model* object (see *Model*).

Boolean views

Boolean view can be declared over several types of variables, and behave as Boolean variables.

bool_not_view

set_bool_view

set_bools_view

Integer views

Integer view can be declared over several types of variables, and behave as Integer variables.

int_offset_view

int_minus_view

int_scale_view

int_abs_view

int_affine_view

int_eq_view

int_ne_view

int_le_view

int_ge_view

Set views

Set view can be declared over several types of variables, and behave as Set variables.

bools_set_view

ints_set_view

set_union_view

set_intersection_view

set_difference_view

graph_node_set_view

graph_successors_set_view

graph_predecessors_set_view

graph_neighbors_set_view

Graph views

node_induced_subgraph_view

edge_induced_subgraph_view

graph_union_view

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search