Unified Programmatic Access to CO Benchmarks, to Connect Constraint Solving Communities

Thomas Sergeys
Department of Computer Science, KU Leuven, Belgium

Ignace Bleukx
Department of Computer Science, KU Leuven, Belgium

Tias Guns
Department of Computer Science, KU Leuven, Belgium

Abstract

Many communities within Combinatorial Optimization (CO) maintain benchmark sets in heterogeneous formats, often tied to specific competitions and solver technologies. Whilst this diversity is of practical and historical importance, it also creates barriers to use and compare methods from different communities. Inspired by the more unified software ecosystem from the ML community, we propose a programmatic abstraction for CO benchmark sets. A unified programmatic interface for downloading, reading and converting datasets across formats. This includes solver-oriented benchmarks such as XCSP3, MIPLib, PB, MaxSATEval, SAT and application-oriented benchmarks such as Nurse rostering, PSPLib (RCSP), and JSPlib. To enable cross-formalism conversions, we provide loaders that bring these dataset instances into CPMpy, a modelling library for constraint programming. CPMpy provides a transformation stack; an extensive set of rewrite operations such as constraint decomposition, linearization, and Boolean encodings, that allow transforming between different constraint formalisms. Based on this, we implement file writers to multiple solver-oriented formats, including MiniZinc, LP file format (ILP), OPB, and DIMACS (W)CNF ((Max)SAT). We demonstrate that this unified abstraction facilitates cross-community access to benchmarks and systematic comparisons of solvers across paradigms.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories; Software and its engineering → Interoperability; Theory of computation → Constraint and logic programming

Keywords and phrases CPMpy, datasets, benchmarking, constraint solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.42

Category Tool Paper

Supplementary Material *Software (Source Code)*: `https://github.com/CPMpy/cpmpy` archived at `swh:1:dir:3c632d72a633f18bc43e8d7d32ee285caaac544b`

Funding This research is partly funded by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (Grant No 101002802, CHAT-Opt). This work was supported by FFG Austria, contract FO999910235.

1 Introduction

The field of combinatorial optimisation (CO) comprises several solving paradigms; (Max)SAT, PB, MIP, CP, etc. Largely driven by solver competitions and paradigm-specific challenges, each community has developed its own modelling languages, benchmark collections and data formats. While this specialization reflects distinct modelling choices and evaluation practices, the fragmentation in file formats creates a barrier towards using and comparing methods from different communities. It complicates systematic comparison across the solving paradigms, as equivalent problem instances are rarely available in multiple, directly

© Thomas Sergeys, Ignace Bleukx, and Tias Guns;
licensed under Creative Commons License CC-BY 4.0

comparable representations. This requires researchers and practitioners to implement parsers and encodings for each benchmark they want to run. To some extent, MiniZinc addresses this issue by offering a solver-independent modelling layer through `.mzn` files. However, many widely used benchmark sets are not (yet) available in this format, and it has limited capability to write other formats, restricting its unifying role. Recent initiatives such as the Global Benchmark Database (GBD)[17] demonstrate the increasing emphasis on standardized benchmark provisioning and metadata management in the SAT and PB communities.

This fragmentation stands in stark contrast to the machine learning (ML) community, which has developed a vast software ecosystem supporting standardized data and model exchange. Centralized dataset hubs such as *Hugging Face Datasets* [1] and *OpenML*[2] allow researchers to share, download and preprocess datasets with a single programmatic call, simplifying reproducibility and reducing the barrier to entry. The Open Neural Network Exchange (ONNX) format provides a framework-independent representation for trained models, enabling interoperability amongst software and hardware platforms [3]. Initiatives like MLCommons' MLPerf provide standardised benchmarks. Standardised metadata formats such as Croissant formalize dataset descriptions, enabling programmatic discovery, versioning, and automated integration into ML pipelines [1]. Together, these developments illustrate how the ML community has prioritised programmatic access, interoperability, and reuse.

Inspired by this ML ecosystem, and motivated by recent calls to consolidate CO benchmarking practices [18] and by discussions at the Dagstuhl workshop on *"Competitions and Empirical Evaluations in Automated Reasoning"* [11], we propose a unified framework for accessing and manipulating commonly used combinatorial optimization benchmarks. In this tool paper, we present a collection of tools that:

1. Support the automatic downloading of benchmarking instances;
2. Provide a Pytorch-compatible dataset interface;
3. Load these diverse instances from file into CPMpy models;
4. Utilise the generic transformations from CPMpy to post instances to a solver; or
5. Write instances to a file in a different format.

Whilst this tooling builds on CPMpy to provide the transformation capabilities, its CO benchmarks can be used in combination with any (constraint modelling) system.

2 Loading benchmarking instances into CPMpy

Inspired by PyTorch's minimal dataset loader, we propose and implement a similar dataset class for CO benchmarks. This dataset object can be used to download, transform, and iterate over problem instances. Note that while our dataset object is inspired by, and compatible with PyTorch-style datasets by design, there is no dependency between our tool and PyTorch.

2.1 Dataset object

To gain programmatic access to various CO benchmarks, we provide a collection of pre-made `Dataset` classes. We distinguish two types of datasets: those that directly represent variables, constraints and an objective in the file format (e.g., FlatZinc, XCSP3, OPB, or DIMACS files), and those that represent data instances from a predefined problem family, without specifying

[1] `https://huggingface.co/docs/datasets`
[2] `https://www.openml.org`

a model (e.g., benchmarks found on PSPLib and schedulingbenchmarks.org). Common logic is shared through an abstract parent class, minimising the work required for adding additional benchmarks. This abstract class is PyTorch compatible, implementing both the `__len__()` and `__get_item__()` functions. It additionally provides class-methods `read()`, to get the raw contents of an instance file, and `_download_file()`, to aid in downloading the benchmark instances (see "download" below). The dataset object and its implementation is entirely independent of any system *using* the dataset object. That is, the dataset object can be used in any other Python library as a light weight iterator over dataset instances.

To create a new dataset, all that must be implemented are 2 methods from Listing 1:

- `download`: retrieve the dataset files, making use of the `_download_file` helper; and
- `open`: define how to open an instance file, especially when files are compressed

For datasets containing only data (without a model), the `parse` method should additionally be implemented to parse the instances into an intermediate data format. E.g., for the RCPSP instances from PSPLib[3], we parse each file into a Pandas DataFrame representing the tasks, their durations, and precedence relations, and into a dictionary for the resource capacities.

Listing 1 The interface needing implementation for new benchmark datasets.

```python
from cpmpy.tools.datasets import FileDataset
class MyDataset(FileDataset):  # torch.utils.data.Dataset compatible

    def __init__( self, root: str=".", transform=None,
                  target_transform=None, download=False, parse=False ):
        # Add additional keyword arguments to distinguish the
        # targeted dataset, like competition year, track, etc.

    def download(self):
        # How and where the data should be downloaded from

    def open(self, instance: os.PathLike) -> io.TextIOBase:
        # Optional: How a problem instance should be opened.

    def parse(self, instance: os.PathLike) -> ...:
        # Optional: parse the data into an intermediate format.
        # Defaults to MyDataset.read(), i.e. the raw file contents.

    def collect_instance_metadata(self, file: os.PathLike) -> dict:
        # Per-instance metadata.

    def categories(self) -> dict:
        # Should return the additional distinguishing keyword arguments.
```

Optionally, implementing `collect_instance_metadata` allows adding domain-specific metadata to each problem instance, while implementing `categories` can provide dataset-wide groupings like competition year, track, etc. These optional methods return the features inside a dictionary, as seen respectively in Listing 2 and Listing 3 for the PSPLib dataset.

Listing 2 PSPLib instance metadata for rcpsp j301_1.

```
{   "num_jobs": 32, "horizon": 158, "duedate": 38, "tardcost": 26,
    "num_renewable_resources": 4, "num_nonrenewable_resources": 0,
    "num_doubly_constrained_resources": 0, "mpm_time": 38,
    "resource_availabilities": [12, 13, 4, 12]   }
```

[3] https://www.om-db.wi.tum.de/psplib/main.html

Listing 3 PSPLib category metadata.

```
{   "variant": "rcpsp", "family": "j30"   }
```

Upon initialization, the dataset is downloaded (if not already present). The instances are downloaded as-is from source and saved locally, without any modification. If an instance is made available as a compressed file, it is stored in compressed form locally as well. Iterating over the dataset results in 2-tuples of the file path of an instance, and a dictionary containing the metadata about that instance (see Listing 4).

Listing 4 Iterating over the 2024 PB competition dataset.

```
from cpmpy.tools.datasets import OPBDataset
for instance_path, metadata in OPBDataset(year=2024, track="DEC-LIN",
    download=True): ...
```

2.2 Data loading (parsing and model building)

Next to the unified access to CO benchmarks, we separately provide utilities for integrating these datasets into CPMpy's modelling framework and transformation architecture.

For datasets directly representing models, we can "load" the constraints and objective 1-on-1 into a CPMpy model, as shown in Listing 5. Iterating over this transformed dataset now yields 2-tuples consisting of the CPMpy model and the metadata dictionary extended with features about the loaded model, like a variable map connecting names to CPMpy variable objects, the number of constraints, etc.

Listing 5 Using the OPB dataset object to download, parse and solve competition instances.

```
from cpmpy.tools.datasets import OPBDataset
from cpmpy.tools.io import load_opb
dataset = OPBDataset(..., transform=load_opb)

for cpm_model, metadata in dataset:
    cpm_model.solve(solver="exact")
    for name, var in metadata["variables"].items():
        print(name, var.value())
```

For datasets representing data only, we first need to "parse" the instance; transforming the file into a solver-agnostic intermediate data format. By setting the `parse=True` constructor argument, instances pass through the dataset's `parse` method. The intermediate format is then used to build a CPMpy model, as shown in Listing 6. Since datasets containing instance data do not explicitly define any constraints, there are many design choice to make regarding the model. The provided modelling contains one set of choices, and can be exchanged for alternatives. In Listing 9 in the appendix, we show the full default RCPSP model as provided.

Listing 6 PSPLib CPMpy model through an intermediate format.

```
# Loading instance to model in two steps; parsing and modelling
from cpmpy.tools.datasets import PSPLibDataset, model_rcpsp
# 1) Parse: dataset has an implementation for "PSPLibDataset.parse()"
dataset = PSPLibDataset(variant="rcpsp", family="j60",
                        download=True, parse=True)

for (tasks, capacities), metadata in dataset:
    # 2) Model: can now be exchanged for any other model implementation
    cpm_model, (start, end, makespan) = model_rcpsp(tasks, capacities)
    cpm_model.solve()

# Or more compact as a transform
```

```
dataset = PSPLibDataset(variant="rcpsp", family="j60", download=True,
                        parse=True, transform=model_rcpsp)
```

3 Constraint reformulations

Solvers from different CO paradigms support different types of decision variables and constraints. Constraint Programming (CP) has the most expressive language, supporting global constraints [25], potentially non-linear, over both integer and Boolean decision variables. Next are Integer Linear Programming (ILP) solvers supporting only linear inequality constraints, Pseudo-Boolean (PB) solvers supporting linear inequality constraints over Boolean variables, and (Max)SAT-solvers which support Boolean formulas in Conjunctive Normal Form (CNF).

Solvers from the more expressive paradigms can take less expressive languages as input, but the reverse direction requires rewriting a set of high-level expressions into a set of lower level expressions, often requiring the use of auxiliary variables. This is achieved through a series of equivalence-preserving constraint reformulations, transforming a constraint model into the target format. We identify five main reformulation stages: decomposition, flattening, linearization, encoding integer variables to Booleans, and encoding pseudo-Boolean constraints into CNF. Each of the transformations is implemented in CPMpy as a separate function.

Decomposition

When a CP solver does not support a particular CP *global constraint*, or when targeting a lower level paradigm, that global constraint needs to be rewritten into a collection of equivalent lower level expressions, i.e. decomposing the constraint. Examples of global constraints include AllDifferent, Element, Cumulative and Circuit.

We partition the constraints resulting from the decomposition into two groups; *value* and *defining*. The *value* constraints represent the logical value of the original constraint. When a global constraint occurs in a reified or nested context (e.g., under implication, disjunction, or negation), these constraints must themselves be reified or nested to preserve compositional semantics [4]. *Defining* constraints define the value of new auxiliary variables introduced by the decomposition, and should always be enforced at top-level. Returning both groups of constraints separately prevents unnecessary reification or nesting. In practice, CPMpy also uses the concept of global constraints to decompose non-linear functions such as Max, Min, Multiplication and Division.

Flattening

The second step in the transformation pipeline is to eliminate *nested* expressions; introducing auxiliary variables such that each constraint is over decision variables rather than over other expressions. To improve the output of this process, we re-use auxiliary variables through common subexpression elimination (CSE). Whenever we introduce an auxiliary variable for a sub-expression, we first check if such a variable already exists in a cache we maintain.

Linearization

To transform the constraint model into a linear or Boolean model, we linearize the global constraints and functions. We employ standard decompositions for most global constraints available in the CPMpy library, and use a few alternative linear-friendly decompositions for some often-used global constraints. These decompositions encode the integer variables into

auxiliary Boolean variables by using a direct encoding. For example, AllDifferent(X) is decomposed into $\sum_{x_i \in X} [\![x_i = v]\!] \leq 1$ for each value in the domain of the variables in X. Similarly, Element(A, idx, v) is decomposed to $\sum_{i=1..|A|} a_i [\![idx = i]\!] = v$.

Additionally, we detect the use of categorical variables and encode them once using a direct encoding, rather than linearizing each $b \Leftrightarrow (x = v)$ expression separately. This is done by checking whether comparison constraints such as $(x = v)$ are used as a subexpression. E.g., the constraint $(x = 3) \vee (y \neq 1)$ is linearized by first adding the direct encoding of both x and y and then linearizing the constraint as $[\![x = 3]\!] + \neg[\![y = 1]\!] \geq 1$, with $[\![x = v]\!]$ the Boolean variable from the direct Boolean encoding.

Boolean encoding of integer variables

To encode Constraint Programming models or ILP models for (Pseudo-)Boolean solvers, all integer variables must be encoded into Boolean variables. We currently support the most common encodings: direct encoding [26], order encoding [24] or log-encoding (also known as binary encoding) [10]. We reuse the direct encoding if it already exists, and otherwise heuristically decide an encoding based on the size of the domain.

CNF encoding of pseudo-Boolean constraints

When targeting a constraint model for (Max)SAT, the last stage in our transformation pipeline consists of encoding all pseudo-Boolean constraints into clausal form. For this final step, we connect to standard tools from the (Max)SAT community, namely PySAT (including pypblib) [15] and the pindakaas library [6], both of which contain a large number of state-of-the-art CNF encodings for pseudo-Boolean constraints.

All the above transformations include encoding choices, e.g. how to decompose the different global constraints, what integer encoding to use, what PB encoding to use etc. These choices are still under active study in the community today to determine different trade-offs related the size of the encoding and the overall runtime performance of the solver. Any library that aims to transform between different formalisms will have to set some reasonable default choices, these can always be changed or improved by expert users.

4 File writing

To convert any CPMpy model back to file, we provide a collection of file format writers. The currently supported formats are MPS, OPB and DIMACS; an overview is given in Table 1. Through the SCIP Python interface we additionally support LP, CIP, GMS, PIP, and MiniZinc files. The file writers make use of the same set of transformations to translate any generic CPMpy model into expressions matching the required output language.

The combination of datasets, loaders, transformations and writers, enables both the benchmarking and translating of dataset across CO paradigms. Starting from a file, a loaded CPMpy model can either be solved directly or can be written back to file in a different format. An example where the XCSP3 dataset is both solved using a SAT-solver, and translated into the DIMACS CNF format is shown in Listing 7.

Listing 7 Solving and writing XCSP3 in SAT.

```
from cpmpy.tools.datasets import XCSP3Dataset
from cpmpy.tools.io import load_xcsp3, write_dimacs
```

Table 1 Initial set of supported datasets and I/O capabilities.

Format	Dataset	File Reading	File Writing
jsplib	JSPLib [19]	✓	
psplib	PSPLib rcpsp [20]	✓	
nurserostering	Nurse rostering (instances 1–24) [23]	✓	
XCSP3	XCSP3 competition [2]	✓	
OPB	PB Competition [22]	✓	✓
MSE (wcnf)	MaxSAT Evaluation [5]	✓	✓
MPS	MIPLib [12]	✓	✓
DIMACS (cnf)	SAT competition [14]	✓	✓

```
dataset = XCSP3Dataset(..., transform=load_xcsp3)

for cpm_model, metadata in dataset:
    # A) Solve the model directly
    cpm_model.solve("pysat")
    # B) or translate to a different format and write to file
    write_dimacs(cpm_model, metadata["name"] + ".cnf")
```

During reformulation of the input model, some variables are encoded and auxiliary variables are introduced. For example, integer decision variables are encoded to Boolean literals when the target format is DIMACS or OPB. To allow for checking validity of the solution found by the solver, we augment the output files with the *semantics* of an auxiliary variable. More specifically, we adopt the VeriPB-style annotations to indicate a literal is part of the encoding of an integer variable. Such annotations allow to reconstruct the value of the original integer variables after solving, and check the solution with the specification of the high-level model.

Listing 8 Different annotations used in DIMACS comment to indicate the meaning of encoding literals for x: direct encoding (left), order encoding (center) and binary encoding (right).

```
p cnf
c 1 x_eq_0
```

```
p cnf
c 2 x_ge_1
```

```
p cnf
c 3 x_bit0
```

5 Experiments

To demonstrate the flexible cross-community evaluations that our work enables, we ran two experiments on five different benchmark sets with six solvers from diverse paradigms;

- **Benchmarks**: XCSP3 (2024 CSP, 2024 COP)[2], PB competition (2024 DEC-LIN, 2024 OPT-LIN)[22] and Nurse rostering (schedulingbenchmarks.org).
- **Solvers**: Z3 (SMT)[7], OR-Tools (CP)[21], Gurobi (ILP)[13], Exact (PB)[9], Pindakaas (SAT)[6], RC2 (MaxSAT)[16]

Experiments were run on Ubuntu 22.04.5 LTS on a single core of an Intel(R) Xeon(R) Silver 4514Y@3.4Ghz. All code is open-source and available as part of the CPMpy constraint modeling library. The results were obtained using the custom branch `benchmark_datasets`[4], our tool will be merged into the main branch of the CPMpy project over time.

First, we show on the selected datasets how many instances can be translated between formats within a time limit of 1200s and memory limit of 16GB. Table 2 shows the number of instances that were successfully converted and the average number of constraints for a

[4] `https://github.com/CPMpy/cpmpy/tree/benchmark_datasets` on commit `cf221b5d`

Table 2 Average model size of successful translations, for some example datasets.

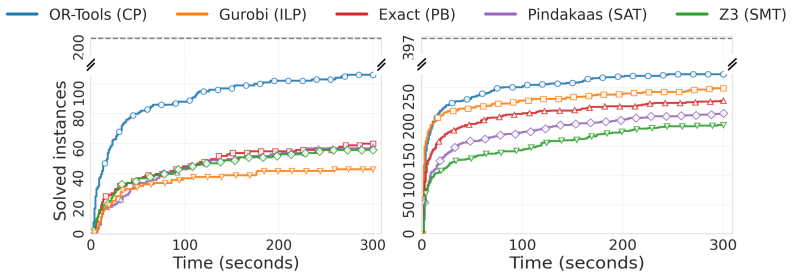
	CP		ILP		PB		(Max)SAT	
Dataset	inst.	avg const.	inst.	avg const.	inst.	avg const.	inst.	avg const.
XCSP3 '24 CSP	200	3 711	147	71 750	156	120 530	138	2 073 629
XCSP3 '24 COP	250	5 571	186	115 895	189	152 628	141	1 2 210 342
PB '24 DEC-LIN	397	101 837	397	101 837	397	101 837	382	494 946
PB '24 OPT-LIN	478	195 006	478	195 006	478	195 006	452	892 742
Nurse rostering	24	1 312 430	18	89 864	19	139 634	17	2 085 722

range of output formats. Unsuccessful conversions were due to memory errors, timeouts and a rare segfault in SCIP's handling of *Abs*. The lower success count for ILP compared to PB can be attributed to the additional memory overhead of writing `.mps` files through the external PySCIPOpt library. The sudden decrease in average number of constraints for Nurse rostering is due to a small number of very large instances which only succeeded for CP.
As a second set of experiments, we posted the models to the six selected solvers and ran with 8GB of RAM per instance and a timeout of 300s. Time measured includes all of parsing, transforming and solving. Solver outputs have been sent through a solution checker, validating that all solvers agree on the satisfiability and optimality of an instance and that all constraints are satisfied. For Gurobi this required setting *IntegralityFocus=1* on XCSP3.
In Figure 1, the solver runtime results for the constraint satisfaction benchmarks (XCSP3 CSP, OPB DEC-LIN) are shown as a set of cumulative distribution plots. The dashed grey line at the top of each plot represents the number of instances in the dataset.
In Figure 2 we show the results from the constraint optimisation benchmarks (XCSP3 COP, OPB OPT-LIN, Nurse rostering). The dashed curves represent the cumulative number of instances proven optimal. The solid curves are a cumulative quality measure that additionally takes the sub-optimal (intermediate) solutions into account. For the latter we adapt the *progress curves* from [8]. Its original formula measures a solver's progression towards its final (sub-)optimal solution as a normalised value within [0,1]. Since we want to compare the relative progress of different solvers, we adapt the measure to instead normalise across a pool of solvers. This results in Equation (1), with s a particular solver, p a problem from the benchmark, t the time, $best_{(s,p,t)}$ the best solution that solver s found for instance i before or at time t, $best_p$ the highest quality solution to p found amongst all solvers at any point in time, similarly for $worst_p$. For some solvers both curves are almost identical, indicating a lacking support for intermediate solutions in either the solver itself or in its CPMpy interface.

$$progress(s,p,t) = \frac{best_{(s,p,t)} - worst_p}{best_p - worst_p} \tag{1}$$

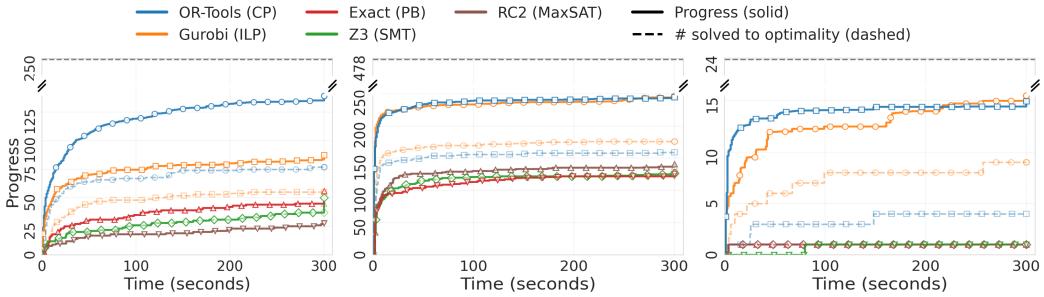
6 Conclusion

In this paper, we introduced a programmatic abstraction of well-known CO benchmarks through a PyTorch-compatible dataset class. Its interface enables downloading, reading, and transforming both solver and application-oriented problem instances. Separately provided loaders can convert an instance into a CPMpy model, giving access to its extensive transformation architecture for rewriting problems to different paradigms.
The proposed work is a step towards unified access to datasets and indirectly solving technologies, to stimulate a more interoperable CO ecosystem. We demonstrated this



(a) XCSP3 2024 CSP **(b)** PB competition 2024 DEC-LIN

Figure 1 Cumulative distribution plots of CSP benchmarks.



(a) XCSP3 competition '24 COP **(b)** PB competition '24 OPT-LIN **(c)** Nurse rostering

Figure 2 Progress curves of COP benchmarks.

capability through a set of cross-community benchmarking runs. Going further, our dataset translation tool can be connected to other initiatives for consolidating benchmarks for the SAT community. For example, the translation capabilities of our tool can be used in the Global Benchmark Database(GBD) [17] as an *instance transformer*. Similarly, the GBD can be configured as a dataset object to gain programmatic access to its instances. For now, we invite the community to use our tool and possibly contribute more datasets, models/transformations, and format writers.

References

1 Mubashara Akhtar, Omar Benjelloun, Costanza Conforti, Luca Foschini, Pieter Gijsbers, Joan Giner-Miguelez, Sujata Goswami, Nitisha Jain, Michalis Karamousadakis, Satyapriya Krishna, Michael Kuchnik, Sylvain Lesage, Quentin Lhoest, Pierre Marcenac, Manil Maskey, Peter Mattson, Luis Oala, Hamidah Oderinwale, Pierre Ruyssen, Tim Santos, Rajat Shinde, Elena Simperl, Arjun Suresh, Goeffry Thomas, Slava Tykhonov, Joaquin Vanschoren, Susheel Varma, Jos van der Velde, Steffen Vogler, Carole-Jean Wu, and Luyao Zhang. Croissant: A metadata format for ml-ready datasets. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 82133–82148. Curran Associates, Inc., 2024. URL: `https://proceedings.neurips.cc/paper_files/paper/2024/file/9547b09b722f2948ff3ddb5d86002bc0-Paper-Datasets_and_Benchmarks_Track.pdf`.

2 Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2024 xcsp3 competition, 2024. `doi:10.48550/arXiv.2412.00117`.

3 Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

4 Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints An Int. J.*, 18(1):1–6, 2013. doi:10.1007/S10601-012-9132-0.

5 Jeremias Berg, Matti Järvisalo, Ruben Martins, Andreas Niskanen, and Tobias Paxian. Maxsat evaluation 2024: Solver and benchmark descriptions, 2024.

6 Hendrik Bierlee and Jip J. Dekker. Pindakaas (pyndakaas-v0.4.1), 2026. doi:10.5281/zenodo.10851855.

7 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

8 David J Eckman, Shane G Henderson, and Sara Shashaani. Evaluating and comparing simulation optimization algorithms. *Under review*, 2021.

9 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/IJCAI.2018/180.

10 Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, pages 1169–1177. Morgan Kaufmann, 1997. URL: http://ijcai.org/Proceedings/97-2/Papers/055.pdf.

11 Johannes Klaus Fichte, Matti Järvisalo, Aina Niemetz, and Guido Tack (organizers). Competitions and empirical evaluations in automated reasoning (dagstuhl seminar 25441). https://www.dagstuhl.de/25441, 2025. Dagstuhl Seminar. URL: https://www.dagstuhl.de/25441.

12 Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-020-00194-3.

13 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

14 Marijn JH Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions, 2024.

15 Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi:10.1007/978-3-319-94144-8_26.

16 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. Rc2: an efficient maxsat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:53–64, September 2019. doi:10.3233/SAT190116.

17 Ashlin Iser and Christoph Jabs. Global benchmark database. In *SAT*, LIPIcs, pages 18:1–18:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.18.

18 Serdar Kadıoglu. Advancing decision science: Lessons from the machine learning community. In *AAAI Constraint Programming and Machine Learning Bridge Program*. Association for the Advancement of Artificial Intelligence, 2025. Position paper.

19 Taemyung Kim. JSPLIB: Job shop scheduling problem library. https://github.com/tamy0612/JSPLIB, 2016.

20 Rainer Kolisch and Arno Sprecher. PSPLIB-a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European journal of operational research*, 96(1):205–216, 1997.

21 Laurent Perron and Frédéric Didier. Cp-sat. no. URL: https://developers.google.com/optimization/cp/cp_solver/.

22 Pseudo-Boolean Competition 2024 (PB24). https://www.cril.univ-artois.fr/PB24/, 2024.

23 Shift Scheduling Benchmark Instances – schedulingbenchmarks.org. https://schedulingbenchmarks.org/. [Accessed 14-05-2026].

24 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009. doi:10.1007/S10601-008-9061-0.

25 Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

26 Toby Walsh. SAT v CSP. In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000. doi:10.1007/3-540-45349-0_32.

A RCPSP CPMpy model

Listing 9 CPMpy model for RCPSP, used in Listing 6.

```python
def model_rcpsp(job_data, capacities):

    model = cp.Model()

    horizon = job_data['duration'].sum()
    makespan = cp.intvar(0, horizon, name="makespan")

    start = cp.intvar(0, horizon, name="start", shape=len(job_data))

    # ensure capacity is not exceeded
    for rescource, capa in capacities.items():
        model += cp.Cumulative(
            start = start,
            duration = job_data['duration'],
            demand = job_data[rescource],
            capacity = capa
        )

    # enforce precedences
    for idx, (jobnr, info) in enumerate(job_data.iterrows()):
        for succ in info['successors']:
            model += end[idx] <= start[succ-1] # job ids start at idx 1

    model += end <= makespan
    model.minimize(makespan)

    return model, (start, end, makespan)
```