Heuristics Selection with ML in CP Optimizer

Hugues Juillé(✉), Renaud Dumeur, and Paul Shaw

IBM France Lab, Orsay, France
{hugues.juille,renaud.dumeur,paul.shaw}@fr.ibm.com

Abstract. IBM® ILOG® CP Optimizer (CPO) is a constraints solver that integrates multiple heuristics with the goal of handling a large diversity of combinatorial and scheduling problems while exposing a simple interface to users. CPO's intent is to enable users to focus on problem modelling while automating the configuration of its optimization engine to solve the problem. For that purpose, CPO proposes an *Auto* search mode which implements a hard-coded logic to configure its search engine based on the runtime environment and some metrics computed on the input problem. This logic is the outcome of a mix of carefully designed rules and fine-tuning using experimental benchmarks. This paper explores the use of Machine Learning (ML) for the off-line configuration of CPO solver based on an analysis of problem instances. This data-driven effort has been motivated by the availability of a proprietary database of diverse benchmark problems that is used to evaluate and document CPO performance before each release. This work also addresses two methodological challenges: the ability of the trained predictive models to robustly generalize to the diverse set of problems that may be encountered in practice, and the integration of this new ML stage in the development workflow of the CPO product. Overall, this work resulted in a speedup improvement of about 14% (resp. 31%) on Combinatorial problems and about 5% (resp. 6%) on Scheduling problems when solving with 4 workers (resp. 8 workers), compared to the regular CPO solver.

Keywords: Combinatorial Optimization · Machine Learning · Lifecycle management

1 Introduction

The search landscape of any combinatorial optimization problem exhibits specific structures. Hence, algorithms that can exploit these structures more efficiently are likely to outperform those that don't. Over the past decades, there has been significant research to determine how to select or adapt search algorithms to improve performance on a specific problem class or problem instance [13]. The algorithm selection problem entails many sub-questions and sub-problems that have been heavily studied. For instance, should a single or a pool of algorithms be

M. Sellmann and K. Tierney (Eds.): LION 2023, LNCS 14286, pp. 208–222, 2023.
https://doi.org/10.1007/978-3-031-44505-7_15

selected, how to distribute compute-time among the selected algorithms, should the algorithm selection be performed before-hand (off-line) or adjusted during search space exploration. Similar questions are raised for computing the features that will characterize a problem class or a problem instance.

While approaches based on hand-crafted heuristics have shown promising results, the design of such heuristics assumes that some internal structures reflecting the intrinsic complexity of problems have been identified and an algorithm can efficiently exploit them. However, because of the ill-defined nature of the search space for hard problems, many of these heuristics are based on rules of thumb rather than computable rigorous mathematical formulations. Therefore, data-driven machine learning methods are natural candidates to address this heuristic design difficulty. The underlying motivation is that ML methods may capture properties of a problem class or a problem instance in a decision model.

Here again, many approaches have been explored for introducing ML for solving combinatorial optimization problems [2,8]. In the more extreme approaches, ML handles problem solving end-to-end by designing new algorithms that integrate trainable models to drive the exploration of the search space. These approaches are usually specific to some problem classes and a frequent goal is to analyze the generalization capability of the trained algorithm to larger problem instances [3,7]. On the other hand, one may choose to capitalize on the large effort that has been put in the design of competitive combinatorial optimization algorithms. In that domain, the two main approaches consist either in using ML before invoking solvers (for instance, by learning how to configure algorithms [9] or by learning some portfolio-based algorithm selection strategy [15]), or in integrating ML in the inner logic of the algorithms to learn rules controlling decision making at runtime [6]. Our approach is the former and, using the taxonomy proposed by [4], can be defined as *ML-in-MH* used to improve the *Algorithm selection* stage.

Objectives and Motivations

The goal of the work presented in this paper is to use Machine Learning to improve the performance of the CPO solver over a large range of application domains and real-world problems (that is, not specific to a problem class). Multiple challenges must be addressed to achieve this goal:

1. **Algorithm selection**: this problem consists in determining how to allocate search heuristics to the available workers (CPU threads) involved in a CPO solve. Predictive models are trained to make this decision, based on an off-line computation of metrics on the input problem.
2. **Training methodology robustness**: Predictive models must not specialize on benchmarks used for training so that CPO solver extended with ML will improve performance over a large class of unseen application domains. Therefore, the training methodology will be designed carefully to limit overfitting issues.

3. **Trained models lifecycle management**: CPO solver is continuously evolving. Existing heuristics efficiency is improved and new heuristics are designed. Therefore, our ML approach must take into account these changes and be integrated in the product development workflow. This means that the training process must be automated as much as possible (e.g., to be ultimately executed from a continuous delivery environment) and reproducible.

The following items summarize the main tasks involved in the implementation of our ML approach:

1. define features to be computed on input problems
2. train a predictive model from these features to drive algorithm selection
3. embed the trained model in CPO executable
4. benchmark this *CPO with ML* version against *regular CPO* version

Section 2 introduces how features are computed based on the formulation of input problems in CPO modelling language. Then, our repository of benchmark problems and our performance assessment process are presented. Section 3 details our formulation of algorithm selection as a ML problem along with the steps involved in training a predictive model and embedding this model in CPO. Robustness and lifecycle management challenges are also addressed in that section. Finally, experimental results for training and benchmarking are presented in Sect. 4.

2 CPO Modelling Language and Features Definition

2.1 CPO Modelling Language

CPO [10] proposes a rich set of constructs to model combinatorial optimization problems. The core of combinatorial problems consists in assigning values to a number of integer decision variables, subject to a number of constraints which enforce conditions on valid domains of these variables. Exploring the problem search space consists in finding valid assignments to its decision variables. A solution to the problem corresponds to a situation where all decision variables are assigned a value while satisfying all constraints. In addition, a measure of quality for solutions (an objective) may be formulated, and the goal is to find a solution that maximize (or minimize) this objective.

CPO supports all regular constraints on integer decision variables: equality, difference, ordering... (low level constraints) along with: alldiff, count, element, distribute, pack... (global constraints)
In addition, CPO introduces the concept of interval variables for the formulation of scheduling problems. An interval is characterized by a start value, an end value, a size and an intensity. Also, interval variables can be optional; that is, one can decide not to consider them in the solution schedule. A number of constructs exploits interval variables to enforce constraints like spanning, precedence, presence, alternative..., etc. Interval variables are also used to define higher level abstractions like sequences of intervals, over which specific constraints may also be enforced.

2.2 Features Definition

A combinatorial problem model can be represented as a directed graph. There is one vertex for each decision variable, one for each constant, one for each constraint and one for each expression. For each constraint (resp. expression), edges connect the associated node with all nodes corresponding to the parameters of the constraint (resp. expression) definition. As a result, vertices with only outgoing edges correspond to decision variables and constants, while vertices with at least one incoming edge correspond to constraints, expressions, or objective definition.

Before actually solving a model, CPO performs a pre-solve stage which consists in a reformulation of the input model. The purpose of this stage is to improve solve performance by removing useless entities or rewriting expressions for faster computation and search space reduction. As a result, this "pre-solved" model can be quite different from the input model and metrics computed on this second model may also vary significantly from those computed on the original model. One will explore whether training the predictive model on one of these constraints models or the other impacts performance.

Our approach to an off-line analysis of input problems consists in engineering a collection of features that captures information about structural properties of the problem. In order to compute features on the graph representing the input problem, the following attributes are defined for each node:

- **Type**: an identifier of the detailed nature of the node (e.g., *integer variable*, *alldiff constraint*...),
- **Flavor**: an identifier that defines a coarser grouping based on the nature of nodes (e.g., *constraint*, *integer expression*...),
- **Constraint**: a flag indicating if the node corresponds to a constraint,
- **Leaf**: a flag indicating if the node has no child (e.g., decision variables, constants...),
- **Root**: a flag indicating if the node has no parent (e.g., objective function, constraints that are not argument of another constraint or expression...),

Using this terminology, Table 1 describes the features that have been explored in this work. This table identifies a group of *mandatory* features that will always be used, while different combinations of features related to search space size will be investigated. In this table, *search space size* refers to the cartesian product of all variables domain. CPO propagation engine reduces the domain of decision variables. As will be seen in the experimental section, considering search space size *before* or *after* this initial propagation (hence resulting in twice as many features) impacts the performance of predictive models.

2.3 Benchmark Problems and Performance Assessment

CPO involves non-deterministic algorithms that are continually tuned to enhance their performances. This tuning needs input from performance tests to be accurate. To enable such testing, the CPO Product Improvement Platform (CPOPIP)

Table 1. Features definition

		Features
"Mandatory" features	**Density-based features**	
	• For each **Type**, density of all nodes of this Type with respect to all other nodes with same value of **Constraint** attribute	158
	• For each **Flavor**, density of all nodes of this Flavor with respect to all other nodes with same value of **Constraint** attribute	41
	Misc ratios based on number of vars, constraints, types	
	Ratio of leaf nodes or constraint nodes over all nodes in graph, ratio of Integer and Interval variables among leaf nodes...	11
	Distribution-based statistics features	
	$log1p()$ of mean, standard deviation and skewness for distributions:	
	• Number of children for *Constraint* nodes	
	• Number of parents for *Leaf* nodes	
	• Number of parents for non-*Leaf* and non-*Constraint* nodes	
	• Number of non-*Leaf* and non-*Constraint* child nodes	12
Log Search Space size features	**Search Space size-based features**	
	First, compute BEFORE and AFTER initial propagation:	
	• For all Integer variables: log of size of domain	
	• For all Interval variables: log of size of interval start domain	
	Then, compute following features:	
	• Sum of logs for Integer variables, Interval variables and overall	3×2
	• For all "density-based" and "misc ratios" features, product of original feature by:	
	- total log search space size over all Integer variables only	210×2
	- total log search space size over all variables	210×2

has been developed and deployed on a dedicated cluster. CPOPIP has been performing benchmarks to monitor enhancements of the CPO product over time. These benchmark problems are based on a repository of a few thousands tests that have been collected from miscellaneous public benchmarks or from real world projects. It is continuously enriched with new tests. All these tests are tagged with attributes to organize them by problem type (integer optimization, scheduling, feasibility) and by family. Problems in a family share some common structure (usually, a family is built from multiple instances of a same problem with varying sizes for decision variables along with customized constraints). At the time of writing of this paper, our repository of tests is composed of 6142 CPO models (2140 combinatorial optimization problems, 483 combinatorial feasibility problems, and 3519 scheduling problems), that are grouped in 327 families. The families size varies from a single instance to 440 for the largest family. CPOPIP embeds multiple tools to support performance analysis at different granularity levels.

A common setup for a benchmarking *campaign* is to execute 10 runs (CPO solve) for each test, with a time limit of 1000 s per run. Latest CPO released version can solve to optimality about 45% of the test for all runs, before the time limit. For less than 2% of these tests, no run can found a first solution. These corresponds to the hardest problems of the repository of tests. For the remaining tests (about 53%), some runs didn't prove optimality but at least one run found a first solution. This corresponds to difficult optimization problems which focus our effort for improving CPO heuristics.

Improvements between CPO versions are assessed by comparing their corresponding benchmarking campaigns. One of the metrics that is computed when comparing two campaigns is the average test speedup. For each model, this indicator evaluates the average ratio of runtimes for the two campaigns to achieve a same performance with respect to the objective value. The speedup value is above 1.0 for tests where the first campaign is faster on average than the second campaign to achieve a same value of the objective (or to find a solution for satisfiability problems), over the 10 runs. The range of values for the test speedup is limited to the interval $[0.01, 100.0]$.

3 General Approach

3.1 Algorithm Selection Problem Formulation

The CP Optimizer search is based on *constructive search*, which is a search technique that attempts to build a solution by fixing decision variables to values. While the built-in CP Optimizer search is based on this technique, the optimizer also uses other heuristics to improve search. These heuristics (or *SearchType* in CPO terminology) are named: *Restart*, *MultiPoint*, *DepthFirst* and *IterativeDiving*.

- *DepthFirst* search is a tree search algorithm such that each fixing, or instantiation, of a decision variable can be thought of as a branch in a search tree. The optimizer works on the subtree of one branch until it has found a solution or has proven that there is no solution in that subtree.
- *Restart* is a depth-first search that is restarted after a certain number of failures that increases after each restart.
- *IterativeDiving* is a search method that attempts to quickly build feasible solutions and improve them using consecutive iterations of backtrack-free search. This heuristics is specialized for Scheduling problems.
- *MultiPoint* search creates a set of solutions and combines the solutions in the set in order to produce better solutions. Multi-point search is more diversified than depth-first or restart search, but it does not necessarily prove optimality or the inexistence of a solution.

Each heuristic can also be manually fine-tuned with specific parameters. *DepthFirst* and *IterativeDiving* are restricted variations of *Restart* heuristics. On the other hand, *Restart* and *MultiPoint* heuristics implement very different

approaches to search. Our experience has shown that, depending on the problem instance, one heuristic can be much faster than the other to solve it. For this reason, we decided to focus on the problem of predicting a score that correlates with the probability that the *MultiPoint* heuristic outperforms the *Restart* heuristic, given a problem instance.

Moreover, when the CPO's *Workers* parameter exceeds 1, multiple worker threads are started when search begins. When the *SearchType* parameter is set to *Auto*, these threads will employ a variety of search heuristics, such as *MultiPoint* and *Restart*. Through the exchange of information, such as intermediate solutions and nogoods, these worker threads can collaborate and cooperate in the search process. Hence, identifying the right mix of heuristics to assign to workers has a strong impact on search performance. In our approach, the predicted score determines the allocation of each heuristic to the different workers, by using a proportionality rule.

This section has formulated the algorithm selection problem as a binary classification problem. Before detailing the ML workflow that will be used to solve this problem in Sect. 3.4, the next two sections introduce how the robustness and life-cycle management challenges have been addressed.

3.2 Training Methodology Robustness

From a Machine Learning methodological point of view, a number of challenges must be overcome. First, even if a significant number of benchmark problems compose the dataset available for training, a few thousand data points very quickly expose us to overfitting and variance issues. Second, the dataset is structured into families that group similar problems together. This raises the issue of diversity in the training data, along with the risk of overfitting to the actual data used for benchmarking and generalizing poorly on unseen problems.

These issues have been mitigated as follows. First, an aggressive splitting strategy has been implemented, using: 30% of problems for training, 30% for validation and 40% for testing. Second, splitting has been performed using stratification based on pairs *(family id, target)*. By using a small fraction of data for training the predictive model and about the same amount of data for validation, overfitting is limited. Indeed, a large validation set enables a more reliable evaluation of generalization so that training is stopped before overfitting occurs. Keeping a large chunk (40%) of all problems for the final evaluation of the predictive model also makes this final measure more reliable. Moreover, stratified splitting introduces more diversity in the training data and improves generalization capability for the trained predictive model.

The LightGBM algorithm [5,11] has been selected for training predictive models. Several reasons motivated our choice of this Gradient Boosted Trees (GBT) framework. First, training models with LightGBM is very fast, which makes heavy Hyper Parameters Optimization (HPO) more manageable to investigate configuration options. Second, this framework exhibits interesting properties for a smooth integration in CPO: serializing decision trees and implementing their evaluation logic is simple (a decision tree is a list of tuples (subtree id, split

feature, threshold value, left and right subtrees id) for internal nodes, and tuples (subtree id, value) for leaf nodes), also the memory footprint of serialized trained models is small.

Also, the logic implemented by decision trees is easy to interpret. Being able to get insights about the main features used to compute class probabilities helps to build confidence in the model and to engineer additional features. Finally, training a LightGBM model is reproducible (given a seed value), which is desirable for our ML workflow.

3.3 Trained Models Lifecycle Management

Heuristics embedded in CPO are continuously improving over time. Therefore, a predictive model trained for a specific version has to be adjusted to reflect the new relative performance of the different heuristics in following versions. The two main challenges to address this particular issue concern: the end-to-end automation of the workflow, and reproducibility.

CPO development and release follow the Continuous Delivery (CD) approach which aims at automating the build, test and release steps of a software. In order to embed a trained predictive model in CPO, all the steps needed to produce this model must be automated in a reliable pipeline. This pipeline covers all the regular ML steps (data preparation, model training, HPO, feature selection...) along with the generation of the actual resources to be packaged in the delivered product. Details about this pipeline are discussed in Sect. 3.4. This pipeline can be executed without any human interaction and can be added to the Continuous Delivery workflow.

Reproducibility is important because the workflow may be executed multiple times for a same version. Therefore, it is important that the outcome of the workflow be identical at each execution. In particular, this means that CPO performance evaluation does not depend on non-deterministic behaviors in the ML workflow. This has been achieved by controlling random seeds of all algorithms involved in the workflow that make non-deterministic decisions: LightGBM, stratified K-fold algorithm, BayesianOptimization [12], features importance assessment algorithm.

3.4 Machine Learning Workflow

This section introduces the different steps involved in the Machine Learning workflow:

- Training dataset preparation:
  - Target definition: For each test problem, compute the speedup between the *MultiPoint* and *Restart* heuristics to assess their relative performance. These speedups result from the comparison of two campaigns performed on our CPOPIP platform. Each campaign executes a single worker that is configured either with *MultiPoint* or *Restart* as the search heuristic. The target for the classification problem corresponds to the winning heuristics

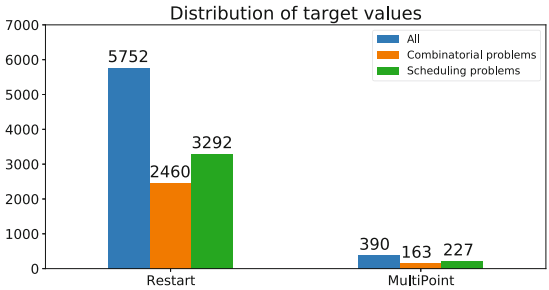

Fig. 1. Number of problems for which each heuristic (*Restart* or *MultiPoint*) outperforms the other, detailed by problem type.

(0 for *Restart*, 1 for *MultiPoint*). The distribution of target values plotted in Fig. 1 illustrates that the two classes are imbalanced. The actual speedup value will be used for weighting training samples.
  - Compute all features listed in Table 1 for all problems in our dataset.
  - Perform the stratified split, using 60% as training/validation dataset and keeping the other 40% aside for testing (as described in Sect. 3.2).
- Model training using LightGBM algorithm:
  - As detailed in Sect. 3.1, algorithm selection has been formulated as a binary classification problem. In that case, cross-entropy is the usual loss function for training.
  - Two-folds cross-validation is used for training. Each fold takes 50% of the input dataset for training (that is 30% of all problems) and the other half for validation. LightGBM training is stopped when no progress is observed for the cross-entropy loss computed on validation data for 50 iterations. The final score of trained models is this out-of-fold (OOF) cross-entropy score.
- Hyper-parameter optimization (HPO): HPO aims at exploring the space of values for the training algorithm parameters in order to identify a configuration for these parameters that optimizes a selected performance metric. At each HPO iteration, 10 runs (changing seed at each run) are executed to account for the randomness incurred by the `feature_fraction` and `bagging_fraction` LightGBM parameters (which control sampling of features and data points). Each run performs a complete two-fold cross-validation training. The OOF cross-entropy loss averaged over all these runs is the actual performance metric that is optimized by HPO. The final output is an assignment of values for selected parameters. In our experiments, HPO is using the BayesianOptimization library [12] and is invoked twice in the Machine Learning workflow:
  1. To identify a good initial configuration for the following list of LightGBM parameters: `max_depth`, `max_bin`, `feature_fraction`, `bagging_fraction`, `bagging_freq`, `min_data_in_leaf`, `lambda_l1` and `lambda_l2`. A detailed documentation of these parameters can be found

in the on-line LightGBM documentation [11]. Features importance will be evaluated based on the configuration of parameters identified after this first HPO round.

2. To select the number of features to keep for training, along with an updated configuration for the above list of LightGBM parameters. Selecting a good subset of features reduces the risk of overfitting and helps generalization.

– Features importance assessment: The purpose of evaluating features importance is to rank features based on their relevance for the training task. We used the *permutation importance* method [1] for ranking features. In the second round of HPO, the number of features to keep in this sorted list is one of the hyper parameter to optimize.
– Selection and serialization of the model to embed in CPO: The LightGBM model with the best OOF score is selected as the final predictive model to be embedded in CPO. This model is serialized as `C++` data structures in a header source file that is added to CPO source code.

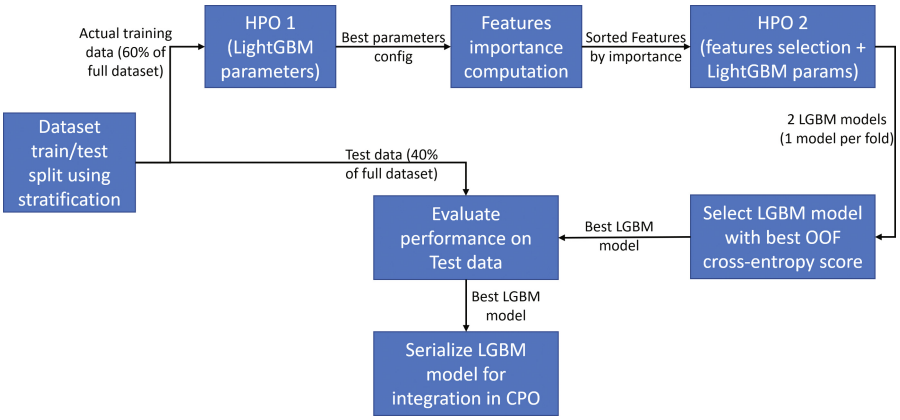

Fig. 2. Training workflow

The output of this Machine Learning workflow is the serialized LightGBM trained model. Figure 2 illustrates how these different steps are sequenced.

3.5 Integration in CPO and Final Performance Evaluation

The CPO solver integrates the features computation code along with the trained model. At runtime, features are computed for the current input problem so that the predictive model can return a score. This score is used as a ratio and drives the strategy to assign heuristics (or *SearchType*) to the different workers involved in search (the number of workers depends on the runtime environment and can be set by the user).

When evaluating the performance of CPO using the predictive model, the overhead introduced by computing features and evaluating prediction for heuristics selection is accounted for in the overall solve time.

The final performance of this problem-specific heuristics selection strategy is assessed by executing two benchmarking campaigns configured with 4 and 8 workers respectively. It is then compared to the benchmarks performed for the regular CPO that performs heuristics selection based on a hard-coded strategy.

4 Experimental Results

4.1 Experimental Setup and Features Sets

In order to explore the relevance of the different categories of features, the configurations detailed in Table 2 will be used in our experiments. All these configurations have a common subset of *mandatory features* that group all features except those related to search space size. As discussed in Sect. 2.2, these features may be computed either on the original model, or on the "pre-solved" model.

Table 2. Configurations for features set

Configuration Id	Description	Features
$C_{\text{Mandatory,Orig}}$	Mandatory features on original model	222
$C_{\text{Mandatory,Presol}}$	Mandatory features on "pre-solved" model	222
$C_{\text{BeforeProp,Orig}}$	Mandatory features + search-space size BEFORE initial propagation on original model	645
$C_{\text{BeforeProp,Presol}}$	Mandatory features + search-space size BEFORE initial propagation on "pre-solved" model	645
$C_{\text{AfterProp,Orig}}$	Mandatory features + search-space size AFTER initial propagation on original model	645
$C_{\text{AfterProp,Presol}}$	Mandatory features + search-space size AFTER initial propagation on "pre-solved" model	645
$C_{\text{Bef\&AftProp,Orig}}$	Mandatory features + search-space size BEFORE and AFTER initial prop. on original model	1068
$C_{\text{Bef\&AftProp,Presol}}$	Mandatory features + search-space size BEFORE and AFTER initial prop. on "pre-solved" model	1068

The next two sections present the experimental results for the training workflow (to identify features and parameters value that result in best performance), and for benchmarking CPO extended with ML.

We will also look at the details of feature importance assessment to identify some of the most relevant features.

4.2 Training Workflow Results

Table 3 summarizes the outcome of the LightGBM training experiments. For each configuration of features set, the number of features optimized by the second round of HPO along with the corresponding averaged cross-entropy loss is reported (at each iteration, HPO performs 10 training runs). Then, using parameters values returned by HPO, 50 additional runs are performed to monitor the evolution of selected metrics during training. In addition to cross-entropy loss, ROC AUC (the area under the receiver operating characteristics curve) is reported for all configurations of features sets. ROC AUC is a metric that provides insights about the sensitivity of binary classifiers to threshold selection for separating positive and negative examples. This information is useful for imbalanced datasets, like in our case.

Table 3. Number of selected features and Out-Of-Fold metrics associated with best parameters value found by HPO, for each configuration of features set.

Configuration Id	Total number of features	Number of selected features	HPO best OOF cross-entropy loss (10 runs)	OOF cross-entropy loss (50 runs)	OOF ROC AUC ROC AUC
$C_{\text{Mandatory,Orig}}$	222	20	0.05641 [8]	0.05641	0.86491
$C_{\text{Mandatory,Presol}}$	222	55	0.05464 [5]	0.05464	0.87390
$C_{\text{BeforeProp,Orig}}$	645	120	0.05387 [2]	0.05387	0.87398
$C_{\text{BeforeProp,Presol}}$	645	296	**0.05369 [1]**	**0.05381**	**0.87847**
$C_{\text{AfterProp,Orig}}$	645	118	0.05415 [3]	0.05415	0.86594
$C_{\text{AfterProp,Presol}}$	645	344	0.05495 [7]	0.05511	0.8803
$C_{\text{Bef\&AftProp,Orig}}$	1068	314	0.05420 [4]	0.05420	0.87384
$C_{\text{Bef\&AftProp,Presol}}$	1068	536	0.05467 [6]	0.05468	0.8754

Table 3 indicates that the best performance is achieved by selecting the top 296 features from the $C_{\text{BeforeProp,Presol}}$ configuration (after sorting by importance). The corresponding trained predictive model is then serialized and embedded in CPO to select heuristics to be used by each worker. Experimental results for this extended CPO version are presented in the next section.

4.3 Benchmarking Results for CPO with ML

A CPO executable embedding the trained predictive model is built and evaluated on our CPOPIP test platform. The baseline for all experiments in this section corresponds to a regular CPO runtime. These regular and *extended with ML* CPO versions differ only in the ML specific logic. All figures reported in this section correspond to solve speedup compared to the baseline.

In order to evaluate performance, a Virtual Best Solver has also been designed by evaluating CPO performance for all possible configurations for assigning

Table 4. Performance of CPO with ML vs Virtual Best Solver **over all problems in benchmark**, by problem type, for 4 and 8 workers

		4 Workers		8 Workers	
		Family geometric av.	All tests geometric av.	Family geometric av.	All tests geometric av.
Combinatorial problems	Virtual Best	1.20	1.23	1.45	1.44
	CPO with ML	**1.08**	**1.11**	**1.28**	**1.29**
Scheduling problems	Virtual Best	1.34	1.27	1.47	1.40
	CPO with ML	**1.09**	**1.06**	**1.12**	**1.07**
Overall	Virtual Best	1.27	1.26	1.46	1.41
	CPO with ML	**1.09**	**1.08**	**1.19**	**1.13**

Restart and *MultiPoint* heuristics to the pool of workers (of size 4 or 8). Then, for each problem, the best speedup among all configurations is kept. The performance of this *Oracle* is then compared to the baseline, providing an upper bound on the best performance that our ML approach can achieve.

Table 5. Performance of CPO with ML vs Virtual Best Solver **for TEST problems only**, by problem type, for 4 and 8 workers

		4 Workers		8 Workers	
		Family geometric av.	All tests geometric av.	Family geometric av.	All tests geometric av.
Combinatorial problems	Virtual Best	1.19	1.24	1.42	1.40
	CPO with ML	**1.13**	**1.14**	**1.30**	**1.31**
Scheduling problems	Virtual Best	1.37	1.27	1.50	1.40
	CPO with ML	**1.08**	**1.05**	**1.09**	**1.06**
Overall	Virtual Best	1.28	1.26	1.46	1.40
	CPO with ML	**1.10**	**1.07**	**1.18**	**1.12**

Tables 4 and 5 present the results, separating Combinatorial and Scheduling problems. The first table presents performance results on the full set of benchmark problems, while only problems from the *Test* dataset (that have not been involved in the training process at all) are considered in the second table. One can notice that the drop in performance between the two setups is minor, which makes us confident in the robustness of the approach (in particular since some families in the test data are not seen at all during training because of their small size).

4.4 Features Importance Analysis

A side question of this work is related to the analysis of the most relevant features that are exploited by trained models. Investigating which metrics computed on

an input problem correlate better with the learning target can help engineering new features and provide some insights to design new heuristics.

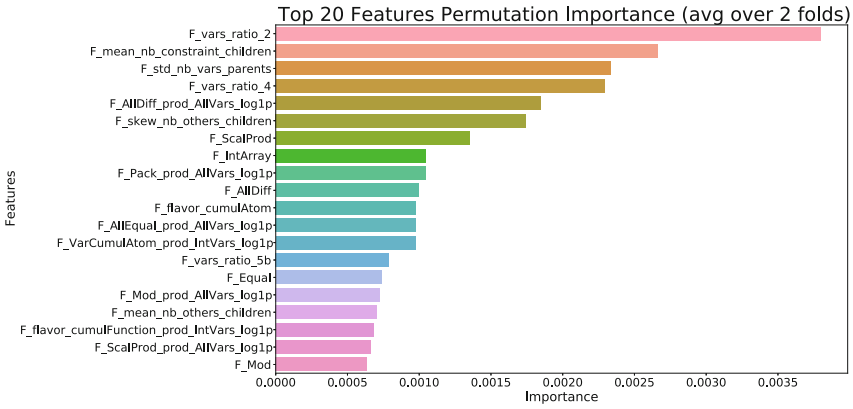

Fig. 3. Top 20 features ranked by permutation importance.

Figure 3 plots the top 20 features ranked by permutation importance for the best predictive model. Interestingly, 5 of the top 6 features in this chart correspond to global structural properties of the directed graph associated with the problem that are independent of constraints types. For instance, the top feature is the proportion of nodes in the graph that are variables nodes, the second one is the mean number of children for constraint nodes...

5 Concluding Remarks

The aim of the work presented in this paper is to extend the CP Optimizer product with some automated heuristics selection capabilities driven by a few hundreds of metrics computed on input problems. A predictive model trained from an extensive repository of benchmark problems supports the heuristics selection strategy. Beyond the challenge of improving the performance of the solver using a Machine Learning approach, embedding a predictive model in a product entails several technical constraints. In particular, the lifecycle of embedded models must be managed so that they are updated when needed. This implies that all decisions involved in the workflow for creating the final predictive model are clearly identified and automated. This involves in particular preparing the training data by benchmarking individual heuristics, assessing and selecting the best configuration of features by performing multiple HPO sessions, or including the serialized final model in the product code.

This work confirms the benefit and the feasibility of using ML methods for improving a productized solver. Our next step is to extend this effort to automatically configure more parameters controlling CPO internal heuristics. Our goal

was a real challenge because of the difficulty to assemble a vast dataset of diverse problems and not limit the domain of application to a few classes of problem. For this reason, adversarial strategies like in [14] are promising approaches that we also intend to investigate in our work.

References

1. Altmann, A., Tolosi, L., Sander, O., Lengauer, T.: Permutation importance: a corrected feature importance measure. Bioinformatics (Oxford, England) **26**, 1340–1347 (2010). https://doi.org/10.1093/bioinformatics/btq134
2. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d'horizon. Eur. J. Oper. Res. **290**, 405–421 (2021)
3. Hottung, A., Tierney, K.: Neural large neighborhood search for the capacitated vehicle routing problem. arXiv abs/1911.09539 (2020)
4. Karimi-Mamaghan, M., Mohammadi, M., Meyer, P., Karimi-Mamaghan, A.M., Talbi, E.G.: Machine learning at the service of meta-heuristics for solving combinatorial optimization problems: a state-of-the-art. Eur. J. Oper. Res. **296**(2), 393–422 (2022)
5. Ke, G., et al.: LightGBM: a highly efficient gradient boosting decision tree. In: Advances in Neural Information Processing Systems, vol. 30, pp. 3146–3154 (2017)
6. Khalil, E.B., Morris, C., Lodi, A.: MIP-GNN: a data-driven framework for guiding combinatorial solvers. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 36, no. 9, pp. 10219–10227 (2022). https://doi.org/10.1609/aaai.v36i9.21262. https://ojs.aaai.org/index.php/AAAI/article/view/21262
7. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: ICLR (2019)
8. Kotary, J., Fioretto, F., Hentenryck, P.V., Wilder, B.: End-to-end constrained optimization learning: a survey. arXiv abs/2103.16378 (2021)
9. Kruber, M., Lübbecke, M.E., Parmentier, A.: Learning when to use a decomposition. In: CPAIOR (2017)
10. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: IBM ILOG CP optimizer for scheduling. Constraints **23**(2), 210–250 (2018). https://doi.org/10.1007/s10601-018-9281-x
11. Microsoft: LightGBM documentation. https://lightgbm.readthedocs.io (2021)
12. Nogueira, F.: Bayesian Optimization: open source constrained global optimization tool for Python (2014). https://github.com/fmfn/BayesianOptimization
13. Smith-Miles, K., Lopes, L.: Measuring instance difficulty for combinatorial optimization problems. Comput. Oper. Res. **39**, 875–889 (2012)
14. Tang, K., Liu, S., Yang, P., Yao, X.: Few-shots parallel algorithm portfolio construction via co-evolution. IEEE Trans. Evol. Comput. **25**(3), 595–607 (2021)
15. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008)