POLITECNICO DI TORINO
Repository ISTITUZIONALE

Learning When to Use Automatic Tabulation in Constraint Model Reformulation

*Original*
Learning When to Use Automatic Tabulation in Constraint Model Reformulation / Cena, C., Akgun, O., Kiziltan, Z., Miguel, I., Nightingale, P., Ulrich-Oltean, F.. - In: IJCAI. - ISSN 1045-0823. - ELETTRONICO. - 2023-August:(2023), pp. 1902-1910. (Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence Macao (CHN) 19-25 August 2023) [10.24963/ijcai.2023/211].

*Availability:*
This version is available at: 11583/2984345 since: 2023-12-07T07:47:15Z

*Publisher:*
International Joint Conferences on Artificial Intelligence

*Published*
DOI:10.24963/ijcai.2023/211

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

(Article begins on next page)

18 November 2026

# Learning When to Use Automatic Tabulation in Constraint Model Reformulation

**Carlo Cena**[1] , **Özgür Akgün**[2] , **Zeynep Kiziltan**[1] , **Ian Miguel**[2] , **Peter Nightingale**[3] and **Felix Ulrich-Oltean**[3]
[1]Department of Computer Science and Engineering, University of Bologna, Italy
[2]School of Computer Science, University of St Andrews, U.K.
[3]Department of Computer Science, University of York, U.K.
carlo98.cena@gmail.com, zeynep.kiziltan@unibo.it, {ozgur.akgun, ijm}@st-andrews.ac.uk,
{peter.nightingale, felix.ulrich-oltean}@york.ac.uk

## Abstract

Combinatorial optimisation has numerous practical applications, such as planning, logistics, or circuit design. Problems such as these can be solved by approaches such as Boolean Satisfiability (SAT) or Constraint Programming (CP). Solver performance is affected significantly by the model chosen to represent a given problem, which has led to the study of model reformulation. One such method is *tabulation*: rewriting the expression of some of the model constraints in terms of a single *table* constraint. Successfully applying this process means identifying expressions amenable to transformation, which has typically been done manually. Recent work introduced an automatic tabulation using a set of hand-designed heuristics to identify constraints to tabulate. However, the performance of these heuristics varies across problem classes and solvers. Recent work has shown learning techniques to be increasingly useful in the context of automatic model reformulation. The goal of this study is to understand whether it is possible to improve the performance of such heuristics, by learning a model to predict whether or not to activate them for a given instance. Experimental results suggest that a random forest classifier is the most robust choice, improving the performance of four different SAT and CP solvers.

## 1 Introduction

Numerous practical applications, such as scheduling, routing, or planning, require us to make a combinatorial decision while respecting a set of constraints, and possibly optimising an objective function. Two of the most common approaches to solving these problems are Boolean Satisfiability (SAT) [Biere *et al.*, 2009] and Constraint Programming (CP) [Rossi *et al.*, 2006]. To apply these solvers one first needs to *model* a problem via a set of decision variables and a set of constraints over those variables that capture valid combinations of decisions. The solver then interleaves a search for a solution in the space of possible assignments with constraint propagation: deduction as to which values can be removed from variables' domains to reduce search.

There exist multiple solvers for each of these approaches, such as CaDiCaL [Biere *et al.*, 2020] and Kissat[1] for SAT, and Minion [Gent *et al.*, 2006] and Chuffed[2] for CP. Modelling languages, such as Minizinc [Nethercote *et al.*, 2007] and Essence Prime [Nightingale, 2022], allow us to write solver-independent models, which are then translated automatically into the format required for the solver chosen.

SAT and CP solvers are sensitive to the model chosen, which impacts solving performance significantly. This has led to the study of model *reformulation* methods: approaches such as *Partial Evaluation* [Lloyd and Shepherdson, 1991], *Common Sub-expression Elimination* (CSE) [Cocke, 1970], *Flattening* and *Tabulation* [Dekker *et al.*, 2017; Akgün *et al.*, 2018; Akgün *et al.*, 2022], are used to simplify the expressions of the model or to rewrite them into ones for which a stronger propagator exists. Some model reformulation methods have been automated, such as generating implied constraints [Rodriguez *et al.*, 2013] that strengthen constraint propagation without losing solutions. Another example is tabulation, partially automated to generate a table from an annotated predicate in MiniZinc [Dekker *et al.*, 2017], or from a set of variables in IBM ILOG CPLEX Optimization Studio[3]. Tabulation can significantly improve an input model because it can increase the strength of solver inference, thus reducing search. However, tabulation does have limitations: (1) its effectiveness depends on the scope and number of possible solutions of the constraints; (2) the tables are time-consuming to create manually and can lead to a less readable model.

Automated tabulation has been used to mitigate, or solve, these problems. Akgün et al. [2018] automatically identify promising constraints to tabulate through a set of hand-designed heuristics, creating the first entirely automatic tabulation method, situated in the constraint modeling tool Savile Row [Nightingale *et al.*, 2017]. They extended their results to new problem classes [Akgün *et al.*, 2022], with a variable effect on solver performance: sometimes improving but sometimes hindering the solution process, either due to the overhead introduced by tabulation or because the generated tables are too large for the table propagator of the considered solver. Machine Learning (ML) techniques have been shown to be

[1]https://github.com/arminbiere/kissat
[2]https://github.com/chuffed/chuffed
[3]https://www.ibm.com/products/ilog-cplex-optimization-studio

increasingly useful in the context of combinatorial decision making and optimization. These approaches have, for example, been used to automatically select the best SAT encoding for a given model [Ulrich-Oltean *et al.*, 2022]. Herein we investigate the use of learning in automatic tabulation, by using a classifier to predict whether or not to use the tabulation heuristics implemented in Savile Row [Akgün *et al.*, 2018].

## 2 Background

Our focus in this paper is on one model reformulation technique, named *automatic tabulation*. We study whether we can improve on its hand-crafted heuristics by adding a machine learning model. First we give the necessary background on tabulation, then give a brief overview of the machine learning methods we use, and finally we explore related work.

### 2.1 Automatic Tabulation

Tabulation is the aggregation of a set of constraint expressions into a single *table* constraint, with the aim of improving the strength of propagation, the efficiency of propagation, or both. A table constraint explicitly lists the allowed tuples of values for the decision variables involved. Table constraints have been intensively researched, they have efficient propagators for CP solvers (e.g. Compact Table and its variations [Demeulenaere *et al.*, 2016]) and their encodings to SAT are well understood. Tabulation has been performed by hand (for example in the Black Hole patience [Gent *et al.*, 2007] and Steel Mill Slab Design [Gargani and Refalo, 2007] problems), demonstrating its utility.

Fully automated tabulation has two steps: identifying candidate sets of constraints, then (for each candidate set) generating the replacement table constraint. The second step is widely automated, for example by Dekker et al. [Dekker *et al.*, 2017], in IBM ILOG CPLEX Optimization Studio [IBM Knowledge Center, 2022], and in ECLiPSe [Le Provost and Wallace, 1992]. Savile Row automates both steps [Akgün *et al.*, 2018; Akgün *et al.*, 2022] using a set of four hand-crafted heuristics to identify promising candidate individual constraints or constraint sets. Each heuristic is based on a known reason for constraints to propagate poorly or otherwise be inefficient when using a propagation-based CP solver. The heuristics are summarised below (full details are given elsewhere [Akgün *et al.*, 2022]).

1. **Identical Scopes** identifies a set of constraints whose scopes are identical;
2. **Duplicate Variables** identifies constraint expressions which contain some variable more than once;
3. **Large AST** identifies constraints where the number of nodes in its AST representation is greater than 5 times the number of distinct decision variables in its scope;
4. **Weak Propagation** identifies constraints that are likely to propagate weakly (in a conventional CP solver) and that share at least one variable with a constraint that propagates strongly.

The Weak Propagation heuristic relies on another heuristic named *GAC estimate* which estimates (for any given constraint expression $c$) whether propagation of $c$ will enforce GAC in a conventional CP solver (defaulting to Minion when behaviour differs between solvers). The four heuristics have been extended to apply to nested constraints (i.e. constraints nested inside operators such as $\vee$ or $\rightarrow$) and numerical expressions, allowing for parts of a top-level constraint to be tabulated when the entire top-level constraint would generate a prohibitively large table. Finally, automatic tabulation has a procedure to convert any Boolean expression to a table constraint, with built-in work limits and progress checks to prevent generation of excessively large tables (whose propagation is likely to be very inefficient) [Akgün *et al.*, 2022]. Automatic tabulation has been shown to work well (in most cases) for CP solvers on a variety of problem classes, but its performance when encoding to SAT is less clear-cut. For both solver classes, there is space to improve upon the accuracy of the hand-crafted heuristics.

### 2.2 Machine Learning with Random Forests

Random forest [Breiman, 2001] is one of the most popular supervised machine learning (ML) methods for several reasons: it is straightforward to apply; it performs well on a diverse set of supervised learning tasks; and it is suitable for both classification and regression tasks. Our task is essentially algorithm selection and random forest is known to perform well in this context. For instance, it is the default classifier used in the sophisticated automated algorithm selection tool AutoFolio [Lindauer *et al.*, 2015]. A random forest model incorporates many decision trees, aggregating their output in some way. When compared to a single decision tree, a random forest is typically more robust to small changes in the input. We use the implementation of random forest provided by the library scikit-learn.[4] This implementation is based on Breiman [2001] except that the individual decision trees predict probabilities for each class and the aggregation step takes an average of the probabilities. Finally the predicted class is the one with the highest average probability.

### 2.3 Related Work

We focus here on applications of ML to constraint model (re)formulation. ML has been successfully used to decide both whether and how to encode constraint models into SAT. Hurley *et al.* [2014] used instance features to make a series of decisions: first whether to use a constraint solver or to encode to SAT, then to choose a combination of SAT encoding scheme and SAT solver. MeSAT [Stojadinović and Marić, 2014] makes decisions about variable encodings as well as some global linear constraints. It is trained on easy problem instances in order to predict good SAT encodings for harder instances. Ulrich-Oltean *et al.* [2022] predicted SAT encodings for pseudo-Boolean and linear constraints using a supervised ML approach (with random forests). They evaluated an existing set of features [Amadini *et al.*, 2014] and a new feature set directly related to pseudo-Boolean and linear constraints. Results show that it is possible to improve on the single best SAT encoding and approach the virtual best, and that the proposed feature set improves on the general instance features for this task. Gent *et al.* [2010] applied decision trees

[4] https://scikit-learn.org/

to identify instances on which *lazy learning* (a conflict learning technique for finite-domain constraint solvers) is expected to be beneficial. Results showed that such a simple classifier is able to reach an accuracy of 99.7% and that the accuracy is almost equal, 99.6%, when only the 3 most important features, from 85, are used. This study, as in [Gent *et al.*, 2010; Ulrich-Oltean *et al.*, 2022], uses a traditional ML model and both general and specific features to improve model reformulation when targeting both SAT and CP. In contrast, our study uses the ML classifiers in combination with the hand-crafted heuristics of [Akgün *et al.*, 2018; Akgün *et al.*, 2022] and we improve the application of the heuristics.

## 3 Methodology

As observed in [Akgün *et al.*, 2022], while automatic tabulation may improve the runtime performance of a solver on one instance, it may worsen it on another. Even if an instance benefits from this method with a solver, changing the solver may lead to inferior results on the same instance. It is non-trivial to know whether activating the heuristics on an instance will improve the solver performance. We here detail our methodology to predict this for a given instance-solver pair.

### 3.1 Overview

We approach this task as a binary classification problem by relying on a simple and an ML-based classifier for each solver, because the table propagator (or encoding) integrated in the solver affects the performance. To train the classifiers, we form a dataset starting from the problem classes and instances used in [Akgün *et al.*, 2022]. We expand the dataset with further problem classes and instances so as to balance the instances benefiting from tabulation with those not.

For an ML-based classifier, an instance should be associated with certain features which are able to describe its relevant characteristics. We construct a set of features and analyze the time required to compute each, so as not to add significant overhead to the model reformulation pipeline. We solve each instance with multiple solvers by applying the heuristics of [Akgün *et al.*, 2018] or not, and record the runtime results. We then create a separate labeled dataset for each solver, by marking each instance with a label indicating that the tabulation is useful or not. The simple and ML-based classifiers are trained and tested on the labeled datasets.

To evaluate the generalization capabilities of the classifiers, we compute the runtime saved by predicting to activate the heuristics, with respect to always activating them. We believe that this metric is more relevant in our task (compared to accuracy, precision, recall, or f1), because it varies across instances, while other metrics do not take this into account. Also, it allows us to compare our results with [Akgün *et al.*, 2018]. Later in Section 4, we will perform a feature selection step for the ML-based classifier where we derive the most relevant features for each solver and remove the less important ones. The purpose is to achieve similar or improved runtime savings while reducing the feature computation overhead.

We identify three different settings in which the classifiers can be trained and tested on a dataset: (i) training and testing on instances of a single problem class, referred to as *per problem class*; (ii) combining all instances across all problem classes into one large set and splitting it into training and test sets, referred to as *by instance*; (iii) training on all instances of all but one problem classes and testing on the latter, referred to as *leave one out*. These are realistic settings suitable to different scenarios. For example, it could be the case that a new instance to be solved belongs to a problem class that has been dealt with many times, or only a small amount of instances may be available for a given problem class, thus making it necessary to use a bigger and diverse dataset. It could also be that a completely new class of instances need to be solved.

In the next subsections, we describe our dataset, features and the classifiers. The code and data are available in github.[5]

### 3.2 Dataset

We define a problem class as a constraint model parametrized by an instance. As there can be multiple models of a given problem, the dataset may contain multiple problem classes referring to the same problem. In our work, a constraint model is expressed in Essence Prime [Nightingale, 2022]. We start with an initial dataset of 40 problem classes used in [Akgün *et al.*, 2022]. This dataset contains many problem classes for which automatic tabulation has been shown to be helpful when using the Minion and Chuffed solvers. To deal with this imbalance, we first generate more instances for the problem classes where tabulation was not beneficial. We then add to this dataset 18 new problem classes, taken mainly from CSPLib[6], that are not expected to benefit from tabulation according to the expert knowledge.

- 2 classes for the **Blocked Queens** problem, a variant of the $n$-**Queens** problem already considered by [Akgün *et al.*, 2022], which contains a set of blocked positions in which a queen cannot be placed.
- 2 classes for the **Plotting** problem [Espasa *et al.*, 2022], which is about removing at least a certain number of coloured blocks from a grid by sequentially shooting blocks into the grid. The difficulty of the problem is related to the complex interactions in it, in fact the blocks may be directly or indirectly affected by each move.
- one class for the **Bombastic** game in which an agent needs to push a few crates to their target position, while moving in a map composed of both normal and ice cells, where the latter transform into dead cells, i.e. obstacles, once walked over.
- 2 classes for **Ramsey Numbers**, where a Ramsey Number *R(m, n)* is the minimum number of vertices such that all undirected simple graphs of order v, i.e. with *v* vertices, contain a clique of order *m* or an independent set of order *n*. Furthermore, *R(m, n)* is also the solution to the party problem in which at least *m* guests know each other or at least *n* don't know each other.
- 3 classes for the **Social Golfers** problem, in which a group of $n = g * m$ golfers needs to play in a golf tournament, with *g* groups of *m* players each, for *w* weeks.

[5] https://github.com/carlo98/ML_AutoTab_CMT

[6] http://www.csplib.org

| Dataset | # Problem classes | # Instances | % Hindered |
|---|---|---|---|
| [Akgün *et al.*, 2022] | 40 | 1410 | 38% |
| New Dataset | 18 | 913 | 37% |
| Overall | 58 | 2323 | 38% |

Table 1: Composition of the dataset.

The goal is to determine the number of unique schedules and creating them if they exist.

- 8 classes for **Quasigroup**, a problem in which the goal is to find out whether there exists an $n \times n$ multiplication table of integers 1..*n*, where each element occurs exactly once in each row and column and certain multiplication axioms hold. The classes considered by us use from 3 to 7 axioms, with those from 3 to 5 further divided in 2 classes with idempotence and not.

We reserve 10 classes of [Akgün *et al.*, 2022] for the test set, due to their small number of instances. Nine of them refer to the problems **Send More Money**, **Tick Tack Toe**, **Tom's Problem**, $n$**-Queens**, **Diet**, **Farm puzzle** and **Grocery**, for which either the maximum number of instances has already been reached or new instances will not add any new information to the dataset. The last problem class in this category is **Magic Square** which is about finding a *n* x *n* matrix containing all the numbers between *1* and $n^2$, with each row, column and main diagonal equal the same sum. To increase the number of instances of this class, it is necessary to increase *n*, but we were not able to solve any instance with $n \geq 7$. For an initial verification of the dataset, we run all the instances using the Minion solver by applying the heuristics of [Akgün *et al.*, 2018] or not. This results in a dataset combining (i) the original 40 problem classes with a total of 1410 instances, of which 38% is hindered by tabulation; (ii) the new 18 classes with a total of 913 instances, of which 37% is hindered by tabulation. The final dataset, as shown in Table 1, is composed of 58 problem classes, 2323 instances of which 38% is hindered by tabulation.

### 3.3 Features

Our feature set contains the features implemented in Savile Row by [Ulrich-Oltean *et al.*, 2022], as they also perform classification at an instance level with the aim of improving the runtime of a given instance and achieve good results on unseen problem classes. In particular, we consider only the set *f2fsr*. In addition, we create four new sets of features, inspired by the heuristics of [Akgün *et al.*, 2018]. The idea is to provide to the classifier information regarding the likelihood of an instance to have (or not) opportunities for tabulation.

- **Overlap**: one feature as the proportion of pairs of constraints in an instance with an overlap of at least 75%.
- **Duplicate variables**: 2 features as the mean and the standard deviation of the number of duplicate variables in the instance constraints.
- **Large AST**: 2 features as the mean and the standard deviation of the AST size of the instance constraints.
- **Strong propagation**: one feature, as the ratio of the instance constraints with strong propagation.

| Feature(s) | Mean (ms) | Std (ms) |
|---|---|---|
| [Ulrich-Oltean *et al.*, 2022] | 191.59 | 503.88 |
| Overlap | 3281.31 | 45454.61 |
| Duplicate variables | 5.29 | 20.54 |
| Large AST | 1.62 | 6.73 |
| Strong propagation | 3.55 | 13.94 |
| Tightness | 5553.45 | 36860.71 |

Table 2: The mean and the standard deviation of the time (in ms) required to compute each set of features on out dataset instances.

The **strong propagation** refers to constraints that are known to have a strong propagator (in a conventional CP solver). While the **large AST** has the same name used by the heuristics, here they represent just the tree size, rather than the comparison between the tree size and a hand-picked threshold, so as to let the ML model to decide how to use the value. We further add the following set of features to our set:

- **tightness**: 2 new features as mean and standard deviation of a value between 0 and 1 that represents the proportion of disallowed tuples in an instance.

The idea is that, a tight instance being difficult to solve is likely to benefit from tabulation. Moreover, this feature has been successfully used in [Gent *et al.*, 2010] for algorithm selection at an instance level using decision trees. To compute it, a certain amount of tuples, starting from the variables' domains need to be generated. We first considered generating 1000 tuples for each constraint, as was done in [Gent *et al.*, 2010], but soon discovered that the runtime overhead to compute the feature for all the constraints of an instance is significant and would risk undermining any possible gain by an ML approach. We therefore experimented with several smaller values and discovered that the tightness value is similar in most cases. For significant time gains, we eventually decided to generate 50 tuples per constraint. Table 2 reports the mean and the st. deviation of the time (in ms) required to compute the features on our instances. As expected, the tightness and overlap are the features with higher computational cost. We will refer to the newly introduced set of features as *hof* (heuristic-oriented features).

### 3.4 Classifiers

Having constructed the dataset and the features, we proceed to create a labeled dataset for each solver. We solve each instance with a solver by applying the heuristics of [Akgün *et al.*, 2018] or not, and record the runtime results as the sum of Savile Row (including tabulation, when applicable) and solver time. We then mark each solver-instance pair with a label *y* such that:

$$y = \begin{cases} 1, & \text{if the heuristics lead to a time gain of at least } \lambda \\ 0, & \text{if the heuristics lead to a time loss of at least } \lambda \end{cases} \quad (1)$$

where $\lambda$ is a user-defined threshold. We discard all the other instances. Given a setting, i.e. a training set and a test set, the labeled datasets are used to train a simple classifier and an ML-based classifier. The simple classifier per solver is trained by summing, over all the instances in the training set, the time saved when using the heuristics with respect to not

| Setting | Solver | Gap |
|---|---|---|
| Per Problem Class | Chuffed | -135 % |
| | Kissat | -328 % |
| | Kissat-MDD | -578 % |
| | Minion | -476 % |
| By Instance | Chuffed | 53 % |
| | Kissat | 56 % |
| | Kissat-MDD | 24 % |
| | Minion | 48 % |
| Leave One Out | Chuffed | 60 % |
| | Kissat | 42 % |
| | Kissat-MDD | 1 % |
| | Minion | 18 % |

Table 3: The gap values for each combination of setting and solver with a random forest classifier. Each value can be at most 100% and gets closer to 100% as the classifier's performances get closer to that of the virtual best classifier which always predicts the correct label. A value $\leq 0\%$ means that the performance is equal to or worse than that of the single best classifier.

using them. If the sum is positive, during testing the classifier activates the heuristics for all the instances, otherwise it never does. We refer to this classifier as the single best classifier. For our ML-based classifier, we use random forest – its advantages are described in Section 2.2. For training a classifier for each solver, we define a custom scoring method for the hyperparameter tuning phase which is detailed in Section 4.1.

# 4 Experimental Study

We have introduced our prediction task as a binary classification problem, described the collection of instances and selected the features as well as the classifiers. In this section we give details of the training and testing phase before going on to present the results and discuss the findings.

## 4.1 Experimental Design

We consider 3 solvers, one of which in two configurations: Minion [Gent *et al.*, 2006], Kissat and Chuffed. Minion is a general-purpose constraint solver, Kissat is a solver which has won recent SAT-solving competitions, and Chuffed is a constraint solver with lazy clause generation. For the Kissat backend Savile Row includes two table encodings, the default encoding is based on [Bacchus, 2007] and the second encoding is based on multi-valued decision diagrams.

We set the time gain threshold $\lambda$ to 1 second, which led to 4 labeled datasets containing 75% of the instances of the original dataset for Minion, 52% for Kissat and Kissat-MDD and 61% for Chuffed. We identified three separate settings for our experiments in Section 3.1. We select an appropriate train-test split ratio for each setting for our empirical evaluation: for the *per problem class* and *by instance* settings we used 90% and 70% of the data for training, respectively. We use a larger proportion of instances for training in the *per problem class* setting because some problem classes do not have many instances (we include problem classes with at least 10 instances in this setting of which at least 2 are hindered and 2 helped by activating the heuristics). Lastly, in *leave one out* we do not define a train-test split ratio, because all the instances of the left-out problem class are used in the test set. We run every instance with the tabulation heuristics switched on and off, with 5 different random seeds so as to mitigate random effects. Each run had a 10 GB memory limit and a 60 minutes time limit. We aggregate the results by computing the mean solving time.

We use a random forest algorithm in all settings to solve our classification problem. To select a subset of the features that has substantial predictive power, we use the *SequentialFeatureSelector* algorithm of scikit-learn on the training set by doing 5 cross-validations with 5 distinct random seeds. We then train both random forest and decision tree classifiers with the selected features. In the runtime of these ML-based classifiers, we include the feature computation time, but exclude the ML training time and inference time, because the former is done once (in a few minutes), while the latter takes a max. of 0.16s for our instances (i.e. well below $\lambda = 1s$).

To choose the best set of hyper-parameters and to weigh the instances during training, a score function $score = \frac{\mathbf{y}_p \cdot \mathbf{s}}{\mathbf{y} \cdot \mathbf{s}}$ is used, where $score$ is clipped between -1 and 1, $\mathbf{y}_p$ is the array of predicted values, $\mathbf{s}$ the time gains associated to each instance and $\mathbf{y}$ the ground truth. The time gains are computed as the difference between the time required to solve an instance without tabulation and the time required to solve the same instance with the heuristics.

When presenting the results, we exclude the outliers from the boxplots, as otherwise it would be difficult to do comparisons due to the significant differences in solving time across some problem classes.

## 4.2 Results

Figure 1 shows the time saved by the two classifiers, with respect to always using the heuristics, for each solver in the 3 settings. The results obtained in the setting *per problem class* show that both classifiers are able to save a significant amount of time by switching on and off the heuristics with respect to always using them, in fact for all solvers the mean is significantly above 0. The single best classifier achieves better results as can be seen by comparing the distributions or, less evidently, from the means. This comparison is more visible in Table 3, where the gap value obtained for each combination of solver and setting is available.

The results reported in Table 3 for the *per problem class* setting show that the random forest classifier performs considerably worse than a single best classifier. This happens because the single best classifier predicts to activate the heuristics based on the total time saved for the instances in the training set and because instances of the same problem class behave in similar ways. However, the gap values are percentages and as can be seen in Figure 1, the differences between the two classifiers are small. For this reason, we avoid investigating the use of subsets of features in this setting. In the case of *by instance* and *leave one out* it can be seen from both Table 3 and Figure 1 that the random forest classifier performs better than the single best classifier. These differences will be further analyzed in Figure 2. *Leave one out* is the hardest setting among all, because the classifier is tested on a problem class which is new, while in the other two settings it is possible or certain that instances belonging to the same problem class appear both in the training and in the test set.

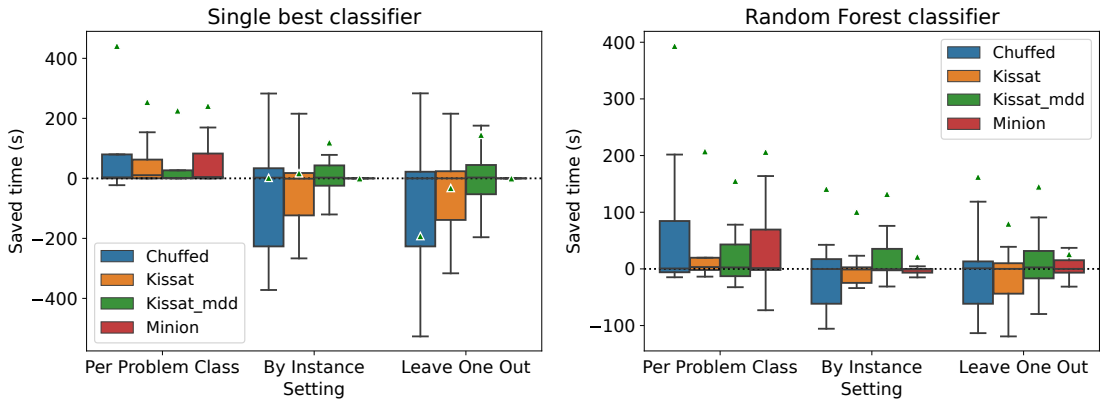


Figure 1: Mean total time saved for each problem class, with respect to always using the heuristics. The single best classifier and a random forest classifier were used. The green triangles represent the mean of each boxplot, while the continuous black lines show its median.

The results show that for all solvers except Kissat-MDD, the ML classifiers save a significant amount of time with respect to the single best classifiers, in some cases closing a large proportion of the gap between single best and virtual best, i.e. *leave one out* with Chuffed and *by instance* with Chuffed and Kissat. In the setting *leave one out* with Kissat-MDD as solver the two classifiers perform approximately in the same way, in fact the gap is 1%.

There are two situations in which the tabulation heuristics and our classifiers agree, and two in which they disagree: (i) the classifier predicts that the heuristics should be used and they select at least one constraint for tabulation; (ii) the classifier switches off the heuristics and the heuristics would have not selected any constraint for tabulation, if activated; (iii) the classifier predicts that the heuristics could aid the solving process, but the heuristics fail to do for all constraints; (iv) the classifier decides that the heuristics should not be used, but they would have selected at least a constraint to tabulate, if activated. For each setting and solver, we also evaluate the number of agreement and disagreement times, along with the time gained by the classifier. Given that we obtain the best results using Chuffed in the *per problem class* setting, we use them as an example in Table 4. As it happens in the majority of cases, the ML model is able to save a significant amount of time by not activating the heuristics, thus avoiding hindering the solving procedure.

By comparing the agreement's time saved with the disagreement's, we find that the latter is always bigger and always positive (for instance, the minimum cumulative time saving is 9,842 seconds in the setting *per problem class* with Kissat-MDD, while the total time saving in this case is 10,712 seconds), while the former can also be negative, meaning that the use of learning to activate the heuristics sometimes leads to worse results with respect to always using the heuristics (for instance, in the setting *leave one out* with Kissat-MDD the disagreements lead to a time loss of 11,100 seconds, while the total time savings is 102,624 seconds).

These results can be explained by looking at how this study works: the classifiers predict whether or not to activate the heuristics and, if the prediction is positive, the heuristics would have run anyway, hence some time is lost by prediction. Moreover, in the case of an ML-based classifier, time is lost in feature computation. Instead, in case of a negative result the classifier can potentially save time, by avoiding the costly computation and failure of the heuristics for each instance's constraints. While in the disagreement case the classifier can avoid situations in which the heuristics would have tabulated a few constraints and hindered the solving procedure, instead of helping it. It could also happen that the classifier decides to switch on the heuristics and that these end up failing in all constraints, but, from what can be seen in the results shown, this happens rarely or with a small impact, especially in the case of the random forest classifier, which seems to be the most robust choice.

| | | RF classifier | |
|---|---|---|---|
| | | Activate | Not Activate |
| Heuristics | Tabulate | 175 (0 s) | 285 (164,748 s) |
| | Not Tabulate | 54 (0 s) | 76 (4 s) |

Table 4: Number of instances and total time saved in seconds (in parenthesis) with respect to always using the heuristics by the RF classifier in different agreement and disagreement scenarios, when using Chuffed in the setting *per problem class*.

## Subset of Features

We search for the most important features to select a subset that has substantial predictive power, to understand whether performance would benefit from the use of a simpler ML model, and to improve model explainability. The employed algorithm selects a different subset of features for each solver, probably because the important characteristics of an instance structure change between solvers. The features selected for the solvers, 6 for Kissat and Chuffed, 7 for Minion and 15 for Kissat-MDD, are in part [Ulrich-Oltean *et al.*, 2022]'s features and in part *hof* features. Minion and Kissat-MDD use features from both sets, while for Kissat and Chuffed the *hof* features do not seem as useful. Minion and Kissat-MDD use strong propagation, large AST, overlap and the standard deviation of the number of duplicate variables.

We train random forest and decision tree classifiers using

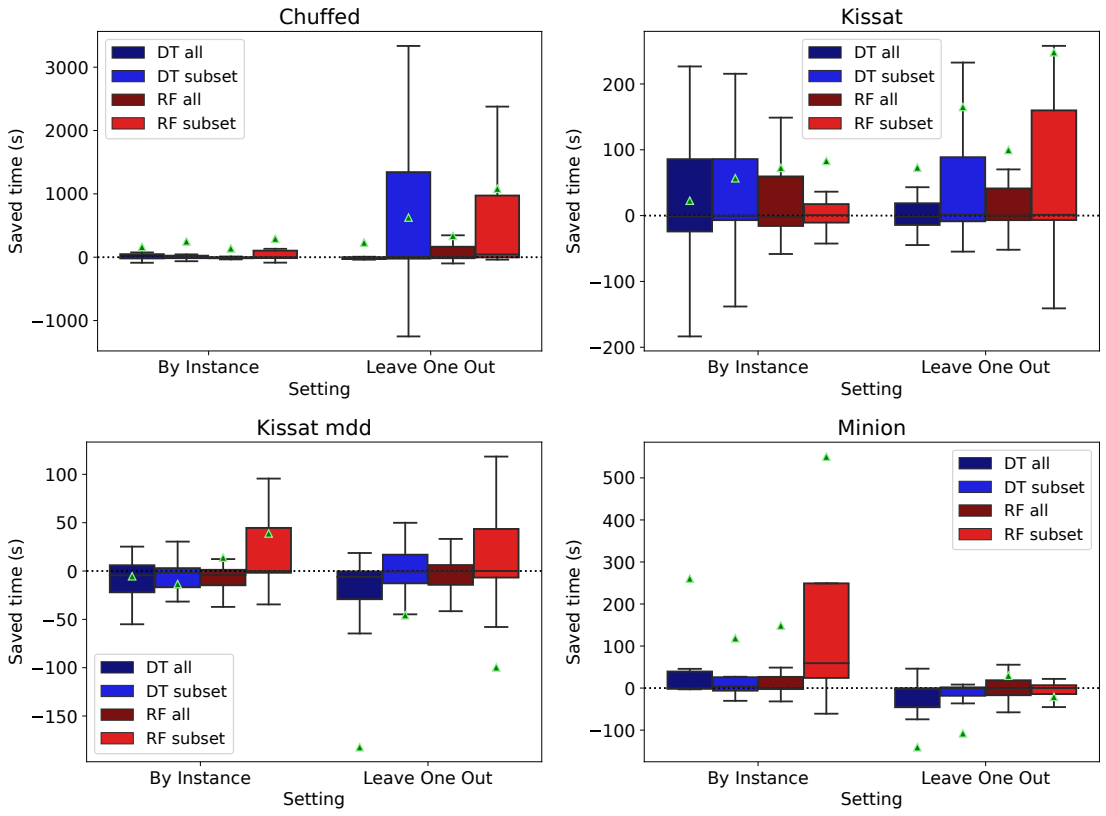


Figure 2: Mean total time saved for each problem class, with respect to the single best classifier, with each solver and different subsets of features. We used a random forest (RF) classifier and a decision tree (DT) classifier.

only the selected features and find the best hyper-parameters using a grid search over the maximum depth of the trees and, in the case of random forests, over the number of decision trees. Figure 2 shows the mean total time saved, for each problem class, with respect to the single best classifier.

In the setting *leave one out* with a subset of the features, it is possible to achieve significantly better results with Chuffed and Kissat, while with Minion the results obtained are slightly worse than those achieved with the entire set of features and with Kissat-MDD the performances of the random forest classifier with a subset of features suffer from negative outliers. Moreover, in *leave one out* with all the other solvers using a decision tree classifier leads to similar or worse results than those of a random forest classifier, which is able to consistently perform better than, or approximately at the same level of, the single best classifier. This can be appreciated by looking at the mean, i.e the green triangle, while the median is always near 0. This is because many instances in the dataset are easy to solve, while others require more time and are not present in the figures shown above, due to the fact that we have decided not to plot the outliers.

Instead, in the setting *by instance* the performances of the random forest classifier with the subsets of features seem to improve significantly compared to those obtained with the complete set of features, even though with Kissat the distribution is more centered towards 0.

## 5 Conclusions

We have shown that by using learning it is possible to improve the performance of the heuristics implemented in Savile Row by [Akgün *et al.*, 2018]. In particular, an hybrid approach which incorporates both ML, in the form of a random forest classifier, and the heuristics of Savile Row is a robust choice to improve, or avoid hindering, the performance of four different solvers in 3 settings, most notably when working on new problem classes. Moreover, this study defines a new set of features, mainly inspired by the automated tabulation heuristics. The analysis of the features shows that it is possible to use just a small subset of them, including some of *hof* features, to achieve comparable, or superior, results to those obtained with the complete set of features.

There are two main avenues of future work. First, it could be interesting to make use of the graph structure of constraint models. GNNs have matured and allow to use our features alongside the graph, thus letting the classifier access features related to the problem/instance structure. Secondly, even though we showed that an ML-based classifier predicting to activate Savile Row's heuristics is in part able to successfully deal with cases in which tabulating a constraint leads to worse performances, it is possible that the current hand-designed heuristics miss some good tabulation opportunities. Therefore it would be interesting to design a ML-based classifier to directly predict candidate sets of constraints to tabulate.

## Acknowledgements

We are grateful for the computational support from the University of York HPC service, Viking and the Research Computing team. This work was supported by EPSRC grants EP/R513386/1, EP/V027182/1 and EP/W001977/1 and by a scholarship from the Department of Computer Science and Engineering of the University of Bologna.

## References

[Akgün *et al.*, 2018] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z. Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming, (CP)*, volume 11008 of *Lecture Notes in Computer Science*, pages 3–12. Springer, Cham, 2018.

[Akgün *et al.*, 2022] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Zeynep Kiziltan, Ian Miguel, P. Nightingale, András Z. Salamon, and Felix Ulrich-Oltean. Automatic tabulation in constraint models. *ArXiv*, abs/2202.13250, 2022.

[Amadini *et al.*, 2014] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1357–1359. Association for Computing Machinery, 2014.

[Bacchus, 2007] Fahiem Bacchus. Gac via unit propagation. In *International conference on principles and practice of constraint programming*, pages 133–147. Springer, 2007.

[Biere *et al.*, 2009] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, first edition, 2009.

[Biere *et al.*, 2020] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. Cadical, kissat, paracooba entering the sat competition 2021. In *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[Breiman, 2001] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[Cocke, 1970] John Cocke. Global common subexpression elimination. *The Association for Computing Machinery Special Interest Group on Programming Languages*, 5(7):20–24, 1970.

[Dekker *et al.*, 2017] Jip J. Dekker, Gustav Björdal, Mats Carlsson, Pierre Flener, and Jean-Noël Monette. Autotabling for subproblem presolving in minizinc. *Constraints*, 22(4):512–529, 2017.

[Demeulenaere *et al.*, 2016] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 207–223. Springer International Publishing, 2016.

[Espasa *et al.*, 2022] Joan Espasa, Ian Miguel, and Mateu Villaret. Plotting: A planning problem with complex transitions. In *International Conference on Principles and Practice of Constraint Programming, (CP)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[Gargani and Refalo, 2007] Antoine Gargani and Philippe Refalo. An efficient model and strategy for the steel mill slab design problem. In *International Conference on Principles and Practice of Constraint Programming*, 2007.

[Gent *et al.*, 2006] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings of the 2006 Conference on Artificial Intelligence*, page 98–102. IOS Press, 2006.

[Gent *et al.*, 2007] Ian P. Gent, Christopher Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M. Smith, and S. Armagan Tarim. Search in the patience game 'black hole'. *AI Communications*, 20(3):211–226, 2007.

[Gent *et al.*, 2010] Ian P. Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil C. A. Moore, Peter Nightingale, and Karen E. Petrie. Learning when to use lazy learning in constraint solving. In *European Conference on Artificial Intelligence, (ECAI)*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 873–878, 2010.

[Hurley *et al.*, 2014] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 301–317. Springer International Publishing, 2014.

[IBM Knowledge Center, 2022] IBM Knowledge Center. The strong constraint, 2022.

[Le Provost and Wallace, 1992] Thierry Le Provost and Mark Wallace. Domain independent propagation. In *Proceedings of FGCS: International Conference on Fifth Generation Computer Systems*, pages 1004–1011. IOS Press, 1992.

[Lindauer *et al.*, 2015] Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

[Lloyd and Shepherdson, 1991] John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3-4):217–242, 1991.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter James Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck,

and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming, (CP)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, Berlin, Heidelberg, 2007.

[Nightingale *et al.*, 2017] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in savile row. *Artificial Intelligence*, 251:35–61, 2017.

[Nightingale, 2022] Peter Nightingale. Savile Row Manual. *ArXiv*, abs/2201.03472, 2022.

[Rodriguez *et al.*, 2013] María Andreína Francísco Rodriguez, Pierre Flener, and Justin Pearson. Generation of implied constraints for automaton-induced decompositions. In *Proceedings of the 2013 IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '13, page 1076–1083. IEEE Computer Society, 2013.

[Rossi *et al.*, 2006] Francesca Rossi, P. V. Beek, and Toby Walsh. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier Science Inc., first edition, 2006.

[Stojadinović and Marić, 2014] Mirko Stojadinović and Filip Marić. meSAT: Multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.

[Ulrich-Oltean *et al.*, 2022] Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Selecting sat encodings for pseudo-boolean and linear integer constraints. In *International Conference on Principles and Practice of Constraint Programming, (CP)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.