Proceedings Paper:

Ulrich-Oltean, Felix, Nightingale, Peter orcid.org/0000-0002-5052-8634 and Walker, James Alfred orcid.org/0000-0003-2174-7173 (Accepted: 2022) Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints. In: 28th International Conference on Principles and Practice of Constraint Programming (CP 2022). LIPICS . (In Press)

Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints

Felix Ulrich-Oltean
Department of Computer Science, University of York, United Kingdom

Peter Nightingale
Department of Computer Science, University of York, United Kingdom

James Alfred Walker
Department of Computer Science, University of York, United Kingdom

Abstract

Many constraint satisfaction and optimisation problems can be solved effectively by encoding them as instances of the Boolean Satisfiability problem (SAT). However, even the simplest types of constraints have many encodings in the literature with widely varying performance, and the problem of selecting suitable encodings for a given problem instance is not trivial. We explore the problem of selecting encodings for pseudo-Boolean and linear constraints using a supervised machine learning approach. We show that it is possible to select encodings effectively using a standard set of features for constraint problems; however we obtain better performance with a new set of features specifically designed for the pseudo-Boolean and linear constraints. In fact, we achieve good results when selecting encodings for unseen problem classes. Our results compare favourably to AutoFolio when using the same feature set. We discuss the relative importance of instance features to the task of selecting the best encodings, and compare several variations of the machine learning method.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint programming, SAT encodings, machine learning, global constraints, pseudo-Boolean constraints, linear constraints

Digital Object Identifier 10.4230/LIPIcs.CP.2022.4

Funding *Felix Ulrich-Oltean*: Supported by grant EP/R513386/1 from the UK Engineering and Physical Sciences Research Council

Acknowledgements We are very grateful to Nguyen Dang for helpful conversations about portfolio approaches. The experiments were undertaken on the Viking Cluster, which is a high performance compute facility provided by the University of York. We are grateful for support from the University of York High Performance Computing service, Viking and the Research Computing team.

1 Introduction

Many constraint satisfaction and optimisation problems can be solved effectively by encoding them as instances of the Boolean Satisfiability problem (SAT). Modern SAT solvers are remarkably effective even with large formulas, and have proven to be competitive with (and often faster than) CP solvers (including those with conflict learning). However, even the simplest types of constraints have many encodings in the literature with widely varying performance, and the problem of predicting suitable encodings is not trivial.

We explore the problem of selecting encodings for constraints of the form $\sum_{i=1}^{n} q_i x_i \diamond k$ where $\diamond \in \{<, \leq, =, \neq, \geq, >\}$, $q_1 \ldots q_n$ are integer coefficients, k is an integer constant and x_i are decision variables. We separate these constraints into two classes: *pseudo-Boolean* (PB) when all x_i are Boolean variables or integer variables with two values; and *linear integer* (LI) when there exists an x_i variable with more than two possible values. We treat these two classes separately, selecting one encoding for each class when encoding an instance.

28th International Conference on Principles and Practice of Constraint Programming (CP 2022).
Editor: Christine Solnon; Article No. 4; pp. 4:1–4:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We select from a set of state-of-the-art encodings, including all four encodings of Bofill et al [8, 9, 7] which are extensions of the Generalized Totalizer [16], Binary Decision Diagram [1], Global Polynomial Watchdog [6], and Sequential Weight Counter [14]. All four of these encodings are for pseudo-Boolean constraints with at-most-one (AMO) sets of terms (where at most one of the corresponding x_i variables are true). The AMO sets come from an integer variable or are detected automatically [5] as described in Section 2.1. We also use Savile Row's *Tree* encoding, which we describe briefly in this paper.

The context for this work is Savile Row [21], a constraint modelling tool that takes the modelling language Essence Prime and can produce output for various types of solver, including CP, SAT, and recently SMT [12]. When encoding a constraint to SAT, two different approaches may be taken depending on the type of constraint. Some constraint types are decomposed into simpler constraints prior to encoding (e.g. allDifferent is decomposed into a set of at-most-one constraints, stating that each relevant domain value appears at most once). Other constraint types are encoded to SAT directly, in which case Savile Row will apply the encoding chosen on the command-line (or the default if no choice is made).

We use a supervised machine learning approach, trained with a corpus of 615 instances from 49 problem classes (constraint models). We show that it is possible to select encodings effectively, approaching the performance of the virtual best encoding (i.e. the best possible choice for each instance), using an existing set of features for constraint problem instances. Also we obtain better performance by adding a new set of features specifically designed for the pseudo-Boolean and linear integer constraints, especially when selecting encodings for unseen problem classes.

1.1 Contributions

In summary, our contributions are as follows:

- We address the problem of selecting SAT encodings for instances of *unseen* problem classes, which we argue is a realistic version of the encoding selection problem. To our knowledge, all previous approaches (such as [15, 24]) train and test their machine learning models on instances drawn from the same set of problem classes.
- We describe a machine learning approach that produces very good results, and that performs much better than the mature, self-tuning algorithm selection tool AutoFolio [19].
- We present a new set of features for pseudo-Boolean and linear integer constraints, and show improved overall performance and robustness when using them.
- We evaluate our machine learning method thoroughly, and present an analysis of feature importance.

1.2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined as a set of variables X, a function that maps each variable to its domain, $D : X \to 2^{\mathbb{Z}}$ where each domain is a finite set, and a set of constraints C. A *constraint* $c \in C$ is a relation over a subset of the variables X. The *scope* of a constraint c, named scope(c), is the set of variables that c constrains. A *constraint optimisation problem* (COP) also minimises or maximises the value of one variable. A *solution* is an assignment to all variables that satisfies all constraints $c \in C$. Boolean Satisfiability (SAT) is a subset of CSP with only Boolean variables and only constraints (*clauses*) of the form $(l_1 \vee \cdots \vee l_k)$ where each l_i is a literal x_j or $\neg x_j$. A *SAT encoding* of a CSP variable x is a set of SAT variables and clauses with exactly one solution for each value in $D(x)$. A SAT encoding of a constraint c is a set of clauses and additional Boolean variables A, where the clauses contain only literals of A and of the encodings of variables in scope(c). An encoding of c has *at least* one solution corresponding to each solution of c. *Generalised arc consistency* (GAC) for a constraint c means that for a given partial assignment, all values are removed from the domain of each variable in scope(c) if they cannot appear in any extended assignment satisfying c. A SAT encoding of c has the property *UP maintains GAC* iff unit propagation of the SAT encoding of c results in the following correspondence: for each variable $x_i \in$ scope(c), the set of solutions of the encoding of x_i corresponds to the set of values in $D(x_i)$ after GAC has been enforced on c.

2 Learning to Choose SAT Encodings

First we describe the palette of encodings for PB and LI constraints, then our approach to selecting encodings using instance features and machine learning.

2.1 SAT Encodings

Recall that we are considering constraints of the form $\sum_{i=1}^{n} q_i x_i \diamond k$ where $\diamond \in \{<, \leq, =, \neq, \geq, >\}$, $q_1 \dots q_n$ are integer coefficients, k is an integer constant and x_i are decision variables (of type integer or Boolean). We use five encodings and each can be applied to either PB or LI constraints, giving 25 configurations in total.

2.1.1 PB(AMO) Encodings

The first four are encodings of PB(AMO) constraints [8, 9, 7], which are pseudo-Boolean constraints with non-intersecting at-most-one (AMO) groups of terms (where at most one of the corresponding x_i variables are true in any solution). Encodings of PB(AMO) constraints can be substantially smaller and more efficient to solve than the corresponding PB constraints [8, 9, 5, 7]. For the four PB(AMO) encodings the constraints must be placed in a normal form where all coefficients are positive, only $\leq$ is allowed, and each x_i must be Boolean. We exactly follow the normalisation rules of Bofill et al [7].

When encoding LI constraints, each integer CSP variable x_i in the constraint (where $|D(x_i)| > 2$) is required to have a direct encoding. When an integer variable (with $|D(x_i)| > 2$) appears in an LI constraint it is replaced with an AMO group of $|D(x_i)|-1$ terms representing each value except the smallest (which is cancelled out). In the case where x_i is an integer variable with two values, Savile Row encodes x_i with a single Boolean variable that is true iff x_i takes its larger value. Also, automatic AMO detection [5] (which applies constraint propagation to find AMO groups among the Boolean terms of the original constraint) is enabled in our experiments. Automatic AMO detection has been shown to substantially improve solving time in some cases [5].

Equality constraints are decomposed into two inequalities ($\leq$) prior to encoding. Disequality constraints and any constraints that are not top-level (i.e. are nested in another expression such as a disjunction) are encoded with the *Tree* encoding, described in Section 2.1.2. Full details of the conversion of integer terms and normalisation for PB(AMO) encodings are given elsewhere [5]. The PB(AMO) encodings are as follows:

MDD The Multi-valued Decision Diagram encoding [8] (a generalisation of the BDD encoding for PB constraints [1]) uses an MDD to encode the PB(AMO) constraint. Each layer of the MDD corresponds to one AMO group. BDDs and MDDs are a popular choice for encoding sums to SAT since they can compress equivalent states in each layer.

GGPW The Generalized Global Polynomial Watchdog encoding [9] (generalising GPW [6]) is based on bit arithmetic and is polynomial in size.

GGT The Generalized Generalized Totalizer [9] encodes the PB(AMO) constraint with a binary tree, where the leaves represent the AMO groups and each internal node represents the sum of all leaves beneath it. GGT compresses equivalent states at its internal nodes. It extends the Generalized Totalizer [16].

GSWC The Generalized Sequential Weight Counter [9] (based on the Sequential Weight Counter [14]) encodes the sum of each prefix sub-sequence of the AMO groups.

Unit propagation on the MDD, GGT, and GSWC encodings enforces GAC on the original constraint c when c is a PB(AMO) $\leq$ constraint [9] but not when c contains integer terms or is an equality. GGPW does not have this property.

2.1.2 Tree Encoding

The *Tree* encoding is related to GGT, however it is not a PB(AMO) encoding. *Tree* does not require the constraint to be an inequality, nor to have positive coefficients.

Tree Given a constraint c, each term is shifted such that its smallest value becomes 0, and k is adjusted accordingly. A binary tree is constructed from the sum, with each term (integer or Boolean) attached to a leaf. Internal nodes represent the sum of the leaves beneath them. The order encoding is required for integer leaf nodes (Savile Row generates the *direct* or *order* encoding for variables as required [20]) and is also used for internal nodes. Each internal node is connected to its two children using the order encoding of linear constraints [25]. The root node represents the entire sum, and its set of values is restricted to those satisfying the constraint c. *Tree* can directly encode constraints with integer terms, equality and disequality, but does not benefit from automatic AMO detection.

Unit propagation on Tree enforces GAC on the original constraint c when c is not an equality or disequality.

The set of 5 encodings is diverse but not exhaustive. Abío et al proposed a BDD-based encoding for linear constraints [3], however it has been directly related to the MDD encoding [10]. In addition to MDD-based encodings, Abío et al propose two further encodings for linear constraints [2]: one based on sorting networks (SN), which is related to the GPW encoding, and another log-based encoding BDD-Dec. Other log encodings such as the one used by Picat-SAT [27] may also be more effective in some cases.

For our experiments we use an extended version of Savile Row 1.9.1 [20]. All constraints other than PB and LI use the default encoding as described in the Savile Row manual.

2.2 Instance Features

f2f We use the `fzn2feat` tool [4] to extract 95 static instance features relating to the number and types of variables and their domains, the types and sizes of constraints and features of the objective in optimisation problems. The full list of features can be found at `https://github.com/CP-Unibo/mzn2feat`; some features were not applicable, e.g. there are no float variables in Essence Prime and Savile Row does not produce all the same annotations.

f2fsr We also re-implement these features as closely as possible in Savile Row, applied to the model directly before encoding to SAT.

lipb We introduce a new set of 45 features describing the PB constraints in a problem instance. We also extract these for LI constraints, giving 90 new features in total. These features are listed in Table 1.

combi We combine the *f2fsr* and *lipb* features.

Table 1 New features for pseudo-Boolean and linear integer constraints. For each aspect of a constraint listed in the left column, we calculate the aggregates in the right column. In the aggregation functions, *IQR* means inter-quartile range, *skew* refers to the non-parametric skew and *ent* is Shannon's entropy. The identifier for each aspect is given in brackets.

Aspect of constraint	Aggregate
Number of (PB or LI) constraints	count
Number of terms (`n`)	min, max, mean, median, IQR, skew, ent, sum
Sum of coefficients (`wsum`)	sum, skew, IQR
Minimum coefficient (`q0`)	min, mean
Maximum coefficient (`q4`)	max, median, mean
Median coefficient (`q2`)	median, skew, ent
IQR of coefficients (`iqr`)	median, skew
Coefficients' quartile skew (`skew`)	mean, min, max, ent
Distinct coefficient values (`sep`)	mean, max
Ratio of distinct coeff. values to number of coeffs. (`sepr`)	mean, max
Number of At-Most-One groups (AMOGs) (`amogs`)	mean
Mean size of AMO group (`asize_mn`)	mean
Mean AMOG size ÷ number of terms (`asize_r2n`)	mean
Mean maximum coeff. size in AMOGs (`amaxw_mn`)	mean
Skew of maximum coeff. in AMOGs (`amaxw_skew`)	mean, ent
Upper limit (k) (`k`)	mean, median, max, IQR, ent, skew
k × number of AMOGs (`k_amo_prod`)	mean, IQR, ent

2.3 Problem Corpus

We use the 65 constraint models with a total of 757 instances from a recent paper [12] in order to work with a wide variety of problem classes. An added advantage is that the models are written in Essence Prime, Savile Row's input language. Unfortunately this collection has a very skewed distribution of instances per problem class, ranging from just 1 to 100. We address this by limiting the number of instances per class to 50 (a random sample) and by adding instances to existing classes where it is easy, such as when instance parameters are just a few integers. We also add two problem classes from recent XCSP3 competitions [18]. More details of the corpus after cleaning are given in Section 3.2 and Table 2.

2.4 Training

We evaluated several classifier models from the `scikit-learn` library [22], including decision trees and forests of extremely randomised trees. We also investigated training various regressors to predict runtime. We find that random forest classifier performs best for our purposes. The `scikit-learn` implementation is based on Breiman's random forests [11], but uses an average of predicted probabilities from its decision trees rather than a simple vote.

We follow the method of Probst et al. [23] who investigated hyperparameter tuning for random forests and concluded that the number of estimators should be set sufficiently high (we use 200) and that it is worth tuning the *number of features*, *maximum tree depth*, and *sample size*. We use randomised search with 50 iterations and 5-fold cross-validation to tune the hyperparameters. We experimented with more tuning iterations but it did not lead to improved prediction quality.

If a classifier makes a poor prediction, the consequences vary. It is possible that the chosen encodings lead to a running time which is very close to that of the ideal choice; the opposite is also true and misclassification can be very expensive. To address this issue, we follow a similar approach to the *pairwise classification* used in AutoFolio [19]: we train a random forest model for each of the possible pairs of encoding configurations. When making predictions, each model chooses between its two candidates. The configuration with most votes is chosen; if two or more configurations have equal votes, we select the one which produced the shortest total running time over the training set. This approach effectively creates a predicted ranking of configurations from the features and leads to better prediction performance than using a single random forest classifier.

To facilitate the pairwise training and prediction approach, we reduce our selection of encoding combinations from 25 (5 PB encodings × 5 LI encodings) to a portfolio of 5, thus needing to train just 10 models (rather than 300 if we had used all 25 choices). We seek to retain performance complementarity as described in [17] from a much reduced portfolio size. The portfolio is built from the timings in the training set using a greedy approach as follows. We begin with a single encoding configuration in the portfolio and then successively add the remaining configuration which would lower the virtual best PAR2 time (PAR2 is defined in Section 3.2) by the biggest margin. We do this until we have a portfolio of 5. We repeat the process using each of the 25 configurations as the starting element and finally select the best-performing portfolio from these 25. Figure 1 shows that this portfolio reduction has a small impact on the virtual best performance across our corpus – the virtual best time for a portfolio of size 5 is within 15% of the time achievable with all configurations.

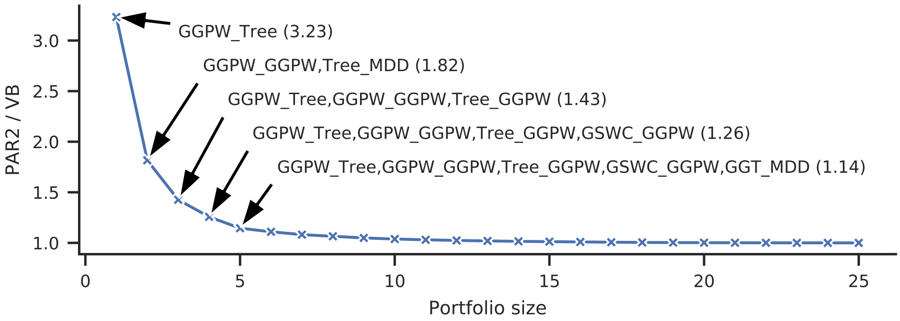

Figure 1 The virtual best PAR2 run-time on our corpus for all portfolio sizes as a multiple of the overall virtual best; the resulting portfolios (of *li_pb* configurations) are shown for sizes 1 to 5.

In addition to the pairwise voting scheme, we implement two further customisations when training the classifiers. Firstly we apply *sample weights* to give more importance to harder instances (those with a longer virtual best runtime) during training. Each instance is given a positive integer weight w according to $w = \lfloor \log_{10}(10(1 + t_{vb}))\rfloor$, where t_{vb} is the virtual best running time for the instance. Secondly, we provide a *custom loss function* for the cross-validation used during hyperparameter tuning. The loss function simply returns the difference in time between the runtime of the chosen encoding configuration and the virtual best for that instance.

To conduct a more complete comparison we also implement two additional alternative setups. Firstly we use a single classifier with a portfolio of 5 configurations (combining PB

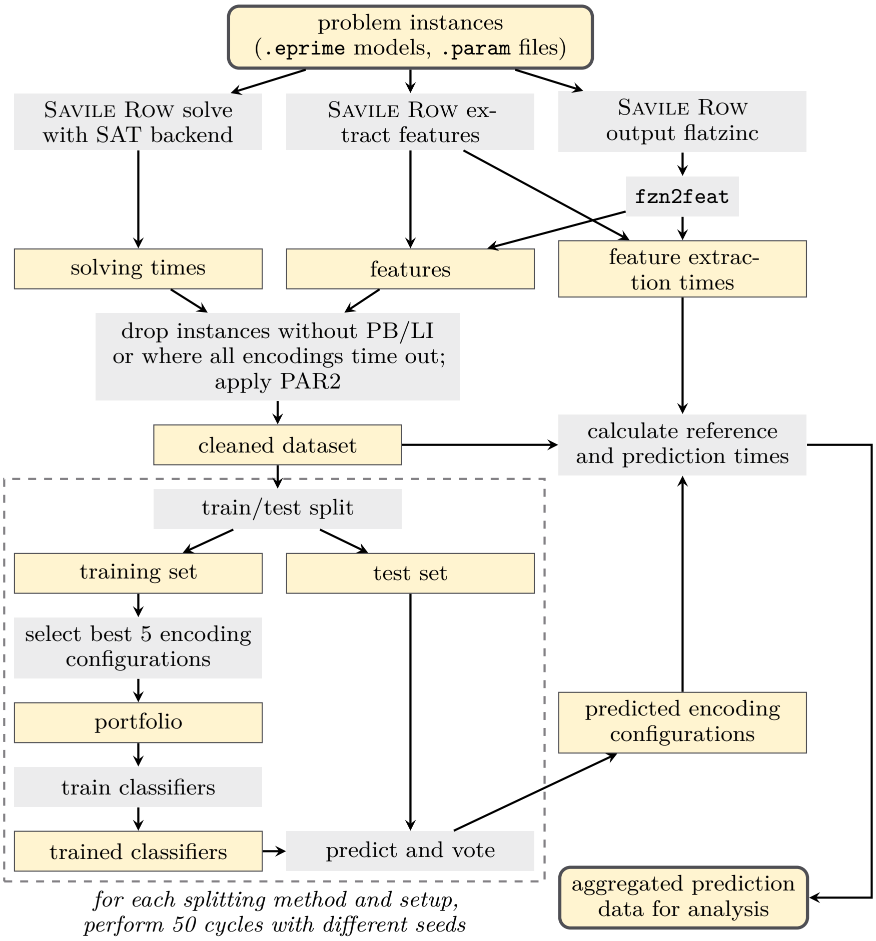
problem instances (.eprime models, .param files)
Savile Row solve with SAT backend
Savile Row extract features
Savile Row output flatzinc
fzn2feat
solving times
features
feature extraction times
drop instances without PB/LI or where all encodings time out; apply PAR2
cleaned dataset
calculate reference and prediction times
train/test split
training set
test set
select best 5 encoding configurations
portfolio
train classifiers
trained classifiers
predict and vote
predicted encoding configurations
aggregated prediction data for analysis
for each splitting method and setup, perform 50 cycles with different seeds

Figure 2 An overview of the steps involved in our experimental investigation. The boxes with solid borders represent data; the grey boxes represent processes.

and LI encodings) and allow it the same number of hyperparameter tuning iterations as the total used by the pairwise classifiers (i.e. 50 × 10). Secondly we modify the pairwise setup to make a separate prediction for PB and LI constraints – this approach has its difficulties because when labelling the dataset with the best encoding for one type of constraint, the other constraint must be chosen somehow. We address this by setting the other constraint to the single best for the training set; however this process is not easily scalable in the same way as the other setups we present. In both of these setups we use sample weighting and the custom loss function for cross-validation.

3 Empirical Investigation

We provide an overview of our experimental process in Figure 2.

3.1 Solving Problem Instances and Extracting Features

We run Savile Row on each instance in the corpus with each of the 25 encoding configurations. The CNF clause limit is set to 5 million and the Savile Row time-out to 1 hour. We switch on automatic detection of At-Most-One constraints [5]. We choose Kissat as our SAT solver

Table 2 Number of instances *(#)*, mean number of PB constaints *(PBs)* and mean number of linear integer constaints *(LIs)* per instance for each problem class in the eventual corpus.

Problem Class	#	PBs	LIs	Problem Class	#	PBs	LIs
killerSudoku2	50	1811.2	129.9	carSequencing	49	435.7	0.0
knights	44	170.5	336.9	langford	39	146.2	0.0
opd	38	21.7	74.8	knapsack	30	1.0	1.0
sonet2	24	10.0	1.0	immigration	23	0.0	1.0
bibd-implied	22	410.6	0.0	handball7	20	705.0	1206.0
mrcpsp-pb	20	90.0	45.7	n_queens	20	1593.0	0.0
efpa	20	156.6	0.0	bibd	19	338.7	0.0
n_queens2	16	309.0	0.0	briansBrain	16	0.0	1.0
life	16	0.0	438.9	molnars	15	0.0	4.0
bpmp	14	14.0	0.0	blackHole	11	202.2	0.0
pegSolitaireTable	8	59.9	0.0	pegSolitaireState	8	59.9	0.0
pegSolitaireAction	8	59.9	0.0	peaceArmyQueens1	7	0.0	1008.0
peaceArmyQueens3	6	0.0	4.0	golomb	6	59.2	38.7
quasiGrp5Idem	6	586.7	0.0	magicSquare	6	118.3	34.0
quasiGrp7	6	410.7	0.0	quasiGrp6	6	410.7	0.0
quasiGrp4NonIdem	4	1067.5	208.0	quasiGrp3NonIdem	4	1067.5	208.0
quasiGrp5NonIdem	4	389.0	0.0	quasiGrp4Idem	4	416.0	208.0
bacp	4	0.0	25.0	quasiGrp3Idem	4	458.0	208.0
waterBucket	4	0.0	46.0	discreteTomography	2	240.0	0.0
solitaire_battleship	2	72.0	16.0	plotting	1	1.0	28.0
nurse	1	27.0	42.0	grocery	1	0.0	2.0
farm_puzzle1	1	0.0	2.0	diet	1	0.0	6.0
sokoban	1	0.0	24.0	sonet	1	3.0	1.0
contrived	1	0.0	4.0	sportsScheduling	1	166.0	64.0
tickTackToe	1	6.0	14.0				

as it formed the basis of the top three performers in the 2021 SAT competition [13]. We use the latest release available at the time, `sc2021-sweep` [26], with default settings and separate time limit of 1 hour. The experiment is run on a research cluster [name removed for anonymity] with Intel Xeon 6138 20-core 2.0 GHz processors; we set the memory limit for each job to 8 GB. We carry out 5 runs (with distinct random seeds) for each configuration to average out stochastic behaviour of the solver.

To extract the features we run each problem instance once with the Savile Row feature extractor and once to generate standard FlatZinc (using the `-flatzinc` flag) followed by `fzn2feat` [4]. We record the time taken to extract the features.

3.2 Cleaning the Dataset

We calculate the median runtime over 5 runs for each instance and encoding configuration, and filter the corpus as follows. We mark a result as timed out if the total runtime (Savile Row + Kissat) exceeds 1 hour. We use PAR2 times, i.e. assigning 2 hours to any result which takes longer than our time-out limit. We choose PAR2 rather than PAR10 as used in some literature [15, 19, 24] because when choosing between our encodings the worst encoding for an instance tends to be around 2 times slower – the median *worst:best* runtime ratio is 1.85 for instances which don't time out. We drop instances if they contain no PB or LI constraints. We exclude instances for which all configurations time out, as well as instances which end up requiring no SAT solving – Savile Row can sometimes solve a problem in pre-processing through its automatic re-formulation and domain filtering. At this point, 615 instances of 49 problem classes remain in the corpus; Table 2 shows the number of instances for each problem class and the mean number of PB and LI constraints per instance.

3.3 Splitting the Corpus, Training and Predicting

For each of our classifier setups and our four featuresets, we run a *split, train, predict* cycle 50 times. We use seeds 1 to 50 to co-ordinate the splits so that we compare the prediction power of the different feature sets and setups using the same training and test sets.

For each cycle, we obtain an 80:20 train to test split using two approaches. The *split-by-instance* approach simply selects instances at random with uniform probability – with this approach, instances of a problem class are usually found in both the training and test sets. The *split-by-class* approach also splits problems randomly but ensures that all instances relating to a problem class end up either in the training or the test set, ensuring that predictions are being made on unseen problem classes. This second method can lead to the test set being slightly larger than 20%.

Prior to training the classifiers, the portfolio of available configurations is built based on the runtimes of the training set. The training instances are labelled for each pairwise classifier with the configuration which has the fastest runtime. For each pairwise classifier, we search the hyperparameter space and fit the model to the training set. Finally, we make predictions using the test set ready for evaluation.

3.4 Evaluating the Performance of Predicted Encodings

To evaluate the impact of using the learnt encoding choices, we calculate two benchmarks commonly used in algorithm selection [17]: the *Virtual Best (VB)* time is the total time taken to solve the instances in a test set if we always made the best choice from our portfolio of configurations; the *Single Best (SB)* time is how long it would take using the one configuration from the portfolio which performs fastest on our training set. In addition we refer to: the time taken using Savile Row's *default (Def)* configuration, which is the *Tree* encoding for both PB and LI constraints, and finally the *Virtual Worst (VW)* time to indicate the overall variation in performance of the encoding configurations in the portfolio.

In Table 3 we report the total PAR2 runtime across all 50 test sets for the predicted encoding configurations from each of the six classifier setups, four feature sets and two splitting methods. The predicted runtimes include the time taken to extract the features.[1] For ease of comparison, we report the runtime as a multiple of the virtual best time. For example, a figure of 2 in Table 3 means that the predictions across the 50 test sets led to a total runtime which was twice as long as the runtime achieved if we always chose the best available encoding combination from each portfolio (as determined from the training set in each cycle).

3.5 Results and Discussion

We found that the machine learning predictors work well, clearly outperforming the *SB* and *Def* configurations. These performance improvements can be achieved with predictions based on the generic CSP feature sets *f2f* and *f2fsr*, but are even better when using the new specialised features (*lipb*). Sometimes the best results are obtained by the combined featureset *combi*. This seems particularly true when predicting for unseen problem classes.

We argue that the split-by-class approach is both a more difficult challenge and closer to a real-world deployment, where a new instance to solve may belong to an unseen problem

[1] For features extracted directly from Savile Row (*f2fsr, lipb, combi*), the feature extraction time added a median of 9% (mean 23%) to the overall running time. The features extracted via `fzn2feat` added 68% (median), 73% (mean).

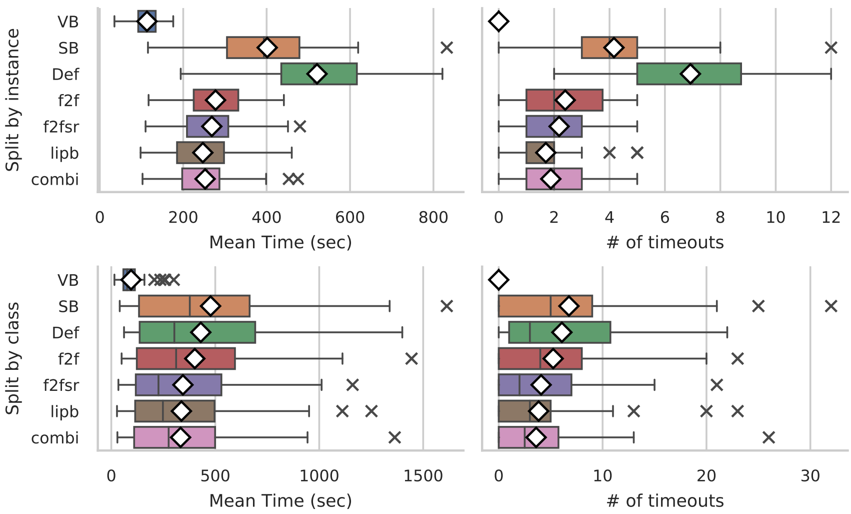

Figure 3 Prediction performance using different featuresets against reference times. We show mean runtime (left) and number of timeouts (right) per test set, when using our preferred setup (*pairwise combined + sample weights + custom loss*).

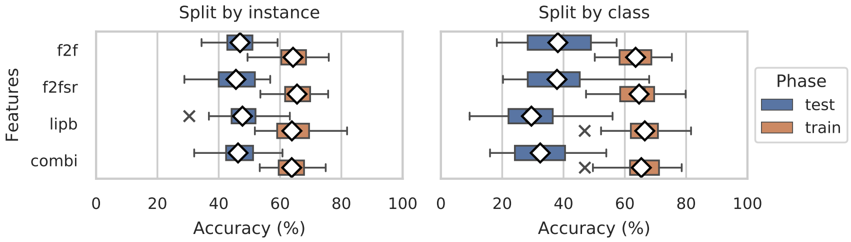

Figure 4 Prediction accuracy per cycle using our preferred setup.

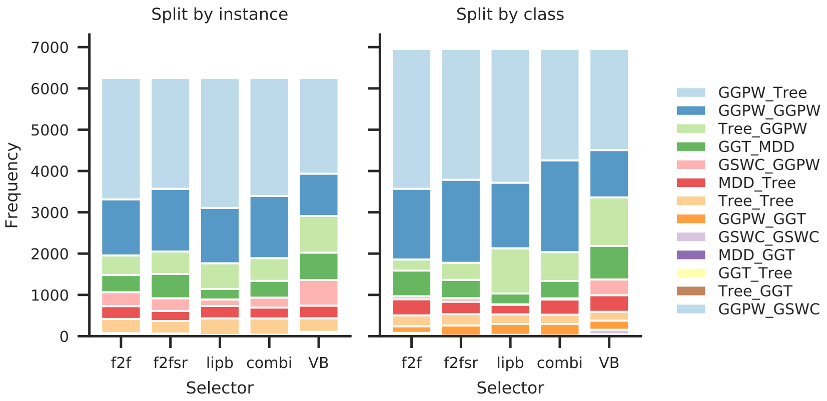

Figure 5 Frequency of each configuration (*li_pb*) selected across the 50 test sets when using each feature set with our preferred setup. We also show the virtual best (VB) configuration for comparison.

Table 3 Total PAR2 times over the 50 test sets as a multiple of the virtual best configuration time. We show the reference times for virtual best (VB), single best (SB), default (Def), and virtual worst (VW) configurations followed by the timings using predictions made using our four feature sets, our six machine learning setups and two splitting strategies. In the setups, *sw* means sample weighting is used and *cl* is when custom loss is used in cross-validation. The best time for each combination of setup, features and split is shown in **bold**. The predicted runtimes include feature extraction time.

Reference Times

Split	VB	SB	Def	VW
by instance	1.00	3.55	4.61	9.75
by class	1.00	5.06	4.53	9.49

Predicted Times

	Split by instance				Split by class			
Setup	f2f	f2fsr	lipb	combi	f2f	f2fsr	lipb	combi
pairwise combined	2.62	2.57	**2.41**	2.51	3.88	3.92	**3.75**	3.90
pairwise combined + sw	2.49	2.46	**2.28**	2.37	3.70	4.12	3.86	**3.52**
pairwise combined + cl	2.62	2.43	**2.36**	2.41	3.97	3.98	**3.58**	3.66
pairwise combined + sw + cl	2.45	2.37	**2.18**	2.23	4.24	3.66	3.56	**3.53**
single combined + sw + cl	2.43	2.43	**2.33**	2.36	4.23	4.43	3.89	**3.74**
pairwise separate + sw + cl	2.35	2.26	2.24	**2.18**	4.01	**3.90**	4.36	3.95

class. However, both approaches are realistic, so we choose *pairwise combined +sw +cl* as our *preferred setup* for the rest of this paper. For split-by-class it is very close to the best, and for split-by-instance is has a sizeable advantage.

In a recent survey, Kerschke et al. state that “State-of-the-art per-instance algorithm selectors for combinatorial problems have demonstrated to close between 25% and 96% of the VBS-SBS gap” [17]. In these terms, our preferred setup using *combi* features closes 38% of the VB-SB gap for unseen classes and 52% for seen classes using the *combi* features (rising to 54% if we use just *lipb* features). Because the distribution of runtimes in the split-by-class trials is skewed, we use a non-parametric statistical test to report on the significance of the improvement achieved by using our classifier. We apply the Wilcoxon Signed-Rank test for paired samples on the mean times from the SB selector choices and our preferred selector using the *combi* features. We obtain a p value of 4.3×10^{-5} (well below even a 1% significance level) and an effect size of -0.67 using the rank-biserial correlation method, or -0.58 using $\frac{z}{\sqrt{n}}$ which would usually be interpreted as a medium to large effect.

Figure 3 summarises the performance for our preferred setup, showing the distribution of mean predicted times per test. The mean values are marked with diamonds and correspond to the numbers reported in Table 3, albeit not scaled. We note that when splitting by instance, the performance across test sets is fairly symmetrical, with very similar means and medians. However, when it comes to splitting by class, the distribution shows positive skew – this likely comes from test sets where there are many instances from an unseen problem class for which the classifier struggles to make the best choices. It is interesting to consider that as we move from the *f2fsr* features to *lipb* and finally to *combi*, the mean remains roughly the same, whereas the median time is increasing. This suggests that the generic *f2fsr* features perform better in the “easier” 50% of test sets, but that good work is undone by costly

Table 4 Comparison of our system's performance with AutoFolio. As before, the overall runtimes for all test instances are reported as multiples of the virtual best (VB) and the best result for each setup is shown in **bold**. The first entry is our system with the preferred setup which includes sample weighting (*sw*) and custom loss (*cl*); to match AutoFolio we use a 10% test split and PAR10 timings. The final two entries show the timings for AutoFolio's predictions after 1 and 2 hours of training. All predicted timings include feature extraction time.

Reference Times

Split	VB	SB	Def	VW
by instance	1.00	10.14	18.60	41.41
by class	1.00	21.91	18.99	43.65

Predicted Times

	Split by instance				Split by class			
Setup	f2f	f2fsr	lipb	combi	f2f	f2fsr	lipb	combi
pairwise combined + sw + cl	5.68	5.95	**5.18**	5.41	14.39	14.58	13.75	**12.45**
AutoFolio (1hr)	20.33	19.90	**19.28**	21.21	21.82	**20.01**	**20.01**	21.87
AutoFolio (2hrs)	20.01	18.79	19.48	**18.33**	22.99	25.19	**17.17**	21.57

misclassification in harder test sets. This observation is also echoed by the distribution of timeouts, also shown in Figure 3. Again, *f2fsr* seems to avoid more timeouts in the easier half of tests; however, when considering the entire distribution, the *lipb* features lead to the fewest timeouts, offering more robust protection against a bad choice of encoding.

A further insight is provided by Figure 4 which shows the accuracy of predictions across the 50 training and test sets – in this figure we see how often the pairwise classifier ends up making exactly the "right" decision. In the split-by-instance scenario the prediction accuracy is fairly consistent across feature sets; however, for unseen classes we see once again that the generic feature sets can spot the very best encoding on more occasions (they have a higher average accuracy on the test sets) but they lead to more costly misclassification as shown by the evaluation of overall runtimes above.

In Figure 5 we show the frequency with which different encoding configurations are predicted. Recall that although we use a portfolio of 5 encodings, this is generated from the training set; consequently the portfolios are different across the 50 sets. One notable finding is that the *GGPW_Tree* and *GGPW_GGPW* appear to be low-risk choices that often perform well, and consequently are favoured by our classifiers. For both split-by-class and split-by-instance, all four selectors choose *GGPW_Tree* or *GGPW_GGPW* more frequently than the VB. Other choices such as *Tree_GGPW* are chosen less often than the VB. Another case in point is the *GSWC_GGPW* encoding: in the split-by-class trials the oracle (VB) uses this configuration in a few hundred cases, but our most successful feature sets (*lipb* and *combi*) eschew it almost entirely. This is likely due to the fact that the GSWC encoding can grow very large and perform badly in some cases; so our predictors seem to choose safer options, being encouraged by the PAR2 penalty to avoid timeouts.

3.6 Comparison with AutoFolio

To further assess the value of our approach, we compare with AutoFolio [19], a sophisticated algorithm selection approach which automatically configures algorithm selectors and "can be applied out-of-the-box to previously unseen algorithm selection scenarios." We use the latest version of AutoFolio (the 2020-03-12 commit which adds a CSV API to the 2.1.2 release) with its default settings. To compare as fairly as possible, we run our system with a similar setup to AutoFolio, namely a 10% test sample and PAR10 times. Our system takes less than 5 minutes to train using 8 cores on the cluster, so we allow AutoFolio 1 hour on one core. We also run it with a more generous budget of 2 hours to see if its performance improves. The results of these runs are shown in Table 4.

Our system's predictions lead to better runtimes than AutoFolio's. AutoFolio is designed to be a good general algorithm selection and configuration system able to make good predictions when choosing between different solvers. It is likely that AutoFolio's sophisticated decision-making is better suited to problems that run much longer or to algorithms for which the likelihood of timeouts or non-termination is more of an issue. It is interesting to note that AutoFolio performs better with the *lipb* features than the generic instance features. Allowing AutoFolio more time for tuning led to marginal improvement with some feature sets, but in some cases actually led to worse performance, for example with *split-by-class* and the *f2fsr* features.

3.7 Feature Importance

We investigate the relative importance of instance features by computing the permutation feature importance. Breiman [11] calculates "variable importance" in random forests by recording the percentage increase in misclassification when each variable (feature) has its values randomly permuted compared to when all features are used. Permuting the values means that the distribution is preserved but the feature effectively becomes noise. This method is applied at prediction time to the test set, unlike the Gini (entropy) feature importance measure which is calculated during training. We implement this analysis but record the mean increase in PAR2 time when each feature is permuted, effectively giving us the extra runtime cost when the feature is lost. Each feature is randomly permuted 5 times and the mean PAR2 time increase recorded. The distribution of feature importance thus calculated is shown in Figure 6. We report on the *lipb* features and on the *combi* feature set which additionally contains the generic features from *f2fsr*.

We can see for both feature sets that the median feature importance in the majority of cases is close to zero, but the mean importance is substantial. This suggests that there are no features which are dominant on their own – most of the time a missing feature incurs no loss of prediction performance. Indeed sometimes removing a feature can improve performance, as shown by some negative costs in most box plots. However, the means of the distributions show that there are cases where each of the features shown is able to prevent a costly wrong choice.

Notice that in the top 20 *combi* features we find a roughly equal mix of generic features and features specific to PB/LI constraints (the names of these features have prefixes `pb_` and `li_`). This is in keeping with the similar performance of the *f2fsr* and *libp* feature sets as shown previously in Table 3.

We suspect that when splitting by instance the system is, to a large extent, recognising problem classes rather than picking out traits of PB/LI constraints. Even when we predict for unseen problem classes, the proportion of PB/LI to generic features in the top 20 is roughly equal. Although we are keeping problem classes apart in this second case, there may be similarities in the constraint models between some problem classes. These similarities might extend beyond the characteristics of PB/LI constraints so the classifier can make a choice of PB/LI encoding on the basis of a choice which worked well in a problem class

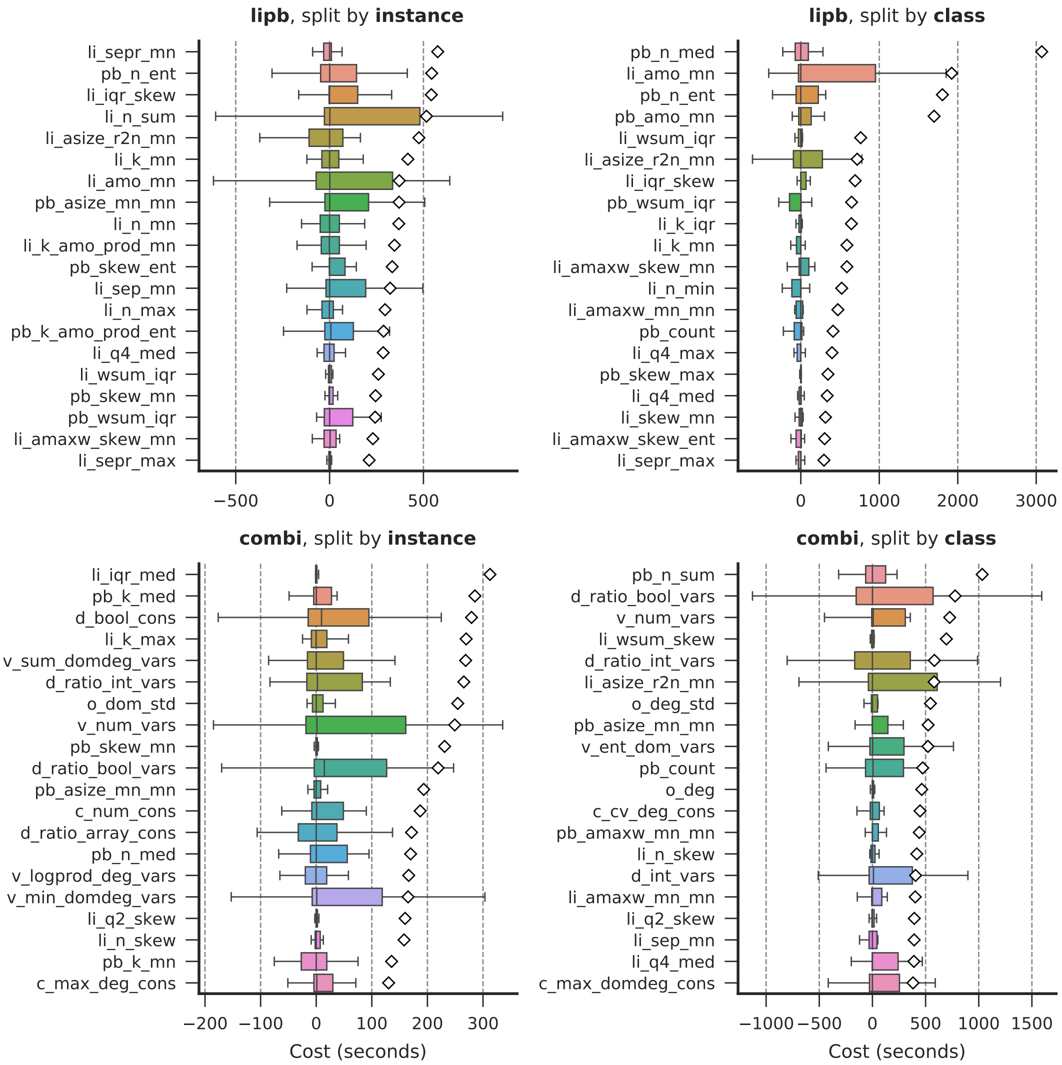

Figure 6 Permutation feature importance: increase in PAR2 time (mean from 5 trials) over 50 *split, train, predict* cycles. We show the top 20 features ordered by mean importance and we do not plot outliers (beyond 1.5 × IQR away from the box). The mean is shown by a diamond. Features beginning `li_` or `pb_` refer to our LI/PB features as introduced in Table 1; the other feature names refer to the generic instance features from the *combi* feature set.

with similar generic features. This interpretation is also supported by the fact that the *lipb* feature set is sometimes matched or even outperformed by *combi* in split-by-class predictions as shown in Table 3.

Of the PB/LI-specific features, the ones extracted from LI constraints feature more strongly – this may reflect the fact that in our corpus the average number of LI constraints per instance is considerably higher than the number of PB constraints, so getting the LI choice right is more important.

There are limitations to how much we can read into the permutation feature importance, especially when we have quite a substantial number of features; a feature may be discriminating but masked by another feature with which it is highly correlated. We have shown that the features in *libp* and *f2fsr* can give comparable prediction performance even though they consider different aspects of a CSP.

4 Related Work

In recent work, new or improved SAT encodings of linear constraints [2] and pseudo-Boolean constraints (combined with AMO constraints) [9, 7] have been devised and their performance compared on several benchmark problems. The scaling properties of encodings are studied, and it is suggested that smaller encodings should be used when coefficients or values of integer variables are large. However, to the best of our knowledge the problem of selecting an encoding (particularly for a previously-unseen problem class) has not been systematically addressed for LI or PB constraints. We use the full set of encodings from one recent paper [9] combined with automatic AMO detection [5].

MeSAT [24] and Proteus [15] both select SAT encodings using machine learning. MeSAT has two encodings of LI constraints: the order encoding [25]; and an encoding based on enumeration of allowed tuples of values (which uses a direct encoding of the CSP variables). It is not clear whether high-arity sums are broken up before encoding. MeSAT selects from three configurations using a k-nearest neighbour classifier using 70 CSP instance features. They report high accuracy (within 4% of the virtual best configuration), however the single best configuration is only 18% slower than the virtual best. Proteus makes a sequence of decisions: whether to use CSP or SAT; the SAT encoding; and the SAT solver to use. The portfolio contains three SAT encodings: direct, support, and a hybrid direct-order, however the encoding of LI constraints is not specified [15]. Proteus generates each candidate SAT encoding and extracts features of the SAT formula to inform its selection – scaling this approach would be difficult when several constraint types are involved, each with many encoding choices. Results show that the choice of encoding (combined with the choice of SAT solver) is important and that machine learning methods can be effective in their context.

5 Conclusions and Future Work

We have shown that it is possible to close much of the performance gap between the single best and virtual best SAT encodings by using machine learning to select encoding configurations based on instance features. We have studied the problem of selecting encodings for instances of previously-unseen classes, a problem that is more challenging and arguably more realistic than the usual setting where training and test instances are drawn from the same set of problem classes. General instance features such as those provided by `fzn2feat` [4] perform well; however the introduction of features specific to linear integer and pseudo-Boolean constraints has enabled us to improve the quality of predictions. We present a machine learning method that performs well, and investigate several variations of it. We have also presented a thorough experimental analysis of the method, a comparison with AutoFolio, and an analysis of feature importance.

We intend to build on these results by considering other constraint types for which multiple SAT encodings exist. It may also be beneficial to expand the problem corpus to have a more even distribution of problem instances per class and to broaden the range of constraint models represented.

References

1 I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. *Journal of Artificial Intelligence Research*,

45:443–480, November 2012. doi:10.1613/jair.3653.

2 Ignasi Abío, Valentin Mayer-Eichberger, and Peter Stuckey. Encoding Linear Constraints into SAT. *arXiv:2005.02073 [cs]*, May 2020. arXiv:2005.02073.

3 Ignasi Abío, Valentin Mayer-Eichberger, and Peter J Stuckey. Encoding linear constraints with implication chains to CNF. In *International Conference on Principles and Practice of Constraint Programming*, pages 3–11. Springer, 2015. doi:10.1007/978-3-319-23219-5_1.

4 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, pages 1357–1359, New York, NY, USA, March 2014. Association for Computing Machinery. doi:10.1145/2554850.2555114.

5 Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z Salamon, Josep Suy, and Mateu Villaret. Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 20–36. Springer, 2019. doi:10.1007/978-3-030-30048-7.

6 Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New Encodings of Pseudo-Boolean Constraints into CNF. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 181–194, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-02777-2_19.

7 Miquel Bofill, Jordi Coll, Peter Nightingale, Josep Suy, Felix Ulrich-Oltean, and Mateu Villaret. SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints. *Artificial Intelligence*, 302:103604, January 2022. doi:10.1016/j.artint.2021.103604.

8 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Compact MDDs for Pseudo-Boolean Constraints with At-Most-One Relations in Resource-Constrained Scheduling Problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 555–562, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2017/78.

9 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. SAT encodings of pseudo-boolean constraints with at-most-one relations. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 112–128. Springer, 2019. doi:10.1007/978-3-030-19212-9.

10 Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. An MDD-based SAT encoding for pseudo-Boolean constraints with at-most-one relations. *Artificial Intelligence Review*, 53(7):5157–5188, 2020. doi:10.1007/s10462-020-09817-6.

11 Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001. doi:10.1023/A:1010933404324.

12 Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. Effective Encodings of Constraint Programming Models to SMT. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 143–159, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-58475-7_9.

13 Marijn Heule, Matti Jarvisalo, Martin Suda, Markus Iser, Tomáš Balyo, and Nils Froleyks. SAT competitions. URL: https://satcompetition.github.io/ [cited 22.02.2022].

14 Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A Compact Encoding of Pseudo-Boolean Constraints into SAT. In Birte Glimm and Antonio Krüger, editors, *KI 2012: Advances in Artificial Intelligence*, Lecture Notes in Computer Science, pages 107–118, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-33347-7_10.

15 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, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-07046-9.

16 Saurabh Joshi, Ruben Martins, and Vasco Manquinho. Generalized Totalizer Encoding for Pseudo-Boolean Constraints. In Gilles Pesant, editor, *Principles and Practice of Constraint*

Programming, Lecture Notes in Computer Science, pages 200–209, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-23219-5_15.

17 Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation*, 27(1):3–45, March 2019. doi:10.1162/evco_a_00242.

18 Christophe Lecoutre and Olivier Roussel. XCSP3 Competition, 2019. URL: http://www.cril.univ-artois.fr/XCSP19/ [cited 22.02.2022].

19 Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. AutoFolio: An Automatically Configured Algorithm Selector. *Journal of Artificial Intelligence Research*, 53:745–778, August 2015. doi:10.1613/jair.4726.

20 Peter Nightingale. Savile Row 1.9.0 Manual. URL: https://savilerow.cs.st-andrews.ac.uk/index.html [cited 22.02.2022].

21 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, October 2017. doi:10.1016/j.artint.2017.07.001.

22 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

23 Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining and Knowledge Discovery*, 9(3):e1301, 2019. doi:10.1002/widm.1301.

24 Mirko Stojadinović and Filip Marić. meSAT: Multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, October 2014. doi:10.1007/s10601-014-9165-7.

25 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, June 2009. doi:10.1007/s10601-008-9061-0.

26 Helsinki Institute for Information Technology University of Helsinki, Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2021 : Solver and Benchmark Descriptions. In *Proceedings of SAT Competition 2021*. Department of Computer Science, University of Helsinki, 2021.

27 Neng-Fa Zhou and Håkan Kjellerstrand. Optimizing SAT encodings for arithmetic constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 671–686. Springer, 2017. doi:10.1007/978-3-319-66158-2_43.