IndiCon: Selecting SAT Encodings for Individual 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

Encoding to SAT and applying a state-of-the-art SAT solver can be a highly effective way of solving constraint problems. For many types of constraints there exist several alternative SAT encodings; and the choice of encoding can significantly affect SAT solver performance for any given problem. Previous work has shown that machine learning (ML) can be used to select SAT encodings for some constraint types, making a choice for each relevant constraint type in a problem instance. The state-of-the-art approach achieves good performance by first building a small portfolio of configurations, then selecting a configuration for a given problem instance using an ML model. The approach necessitates generating training data for every combination of encodings for the constraint types, thus it scales exponentially as more constraint types are added. In this work, we select potentially different encodings for each individual constraint in a problem instance. We are able to match the state-of-the-art performance while avoiding any limitation on the number of constraint types considered. To achieve this we are proposing new individual constraint features, we present a novel method for generating training data, and we have developed a new machine learning pipeline involving both unsupervised and supervised learning.

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

Funding *Felix Ulrich-Oltean*: Supported by grants EP/R513386/1 and EP/W001977/1 from the UK Engineering and Physical Sciences Research Council
Peter Nightingale: Supported by grant EP/W001977/1 from the UK Engineering and Physical Sciences Research Council

Acknowledgements The experiments for this paper were conducted on the Viking cluster, which is a high performance compute facility provided by the University of York. We are grateful for computational support from the University of York IT Services and the Research IT team.

1 Introduction

A popular and effective way of solving constraint satisfaction and optimisation problems (CSPs and COPs) is by reformulating them as instances of the Boolean satisfiability problem (SAT). This process, usually known as *encoding* can take into account the particular structure of different constraint types in the constraint modelling language. For many constraint types, a variety of SAT encodings exist, i.e. schemes for representing a constraint in a CSP as a set of Boolean variables and a propositional formula over those variables.

The task of automatically selecting suitable encodings for constraints into SAT has been addressed previously [3, 6, 9]. Recent work shows that machine learning (ML) models can be trained to select a good encoding for pseudo-Boolean (PB) and linear integer (LI) constraints [9]. The ML-based selections lead to significant performance improvements over

the single best encoding (based on the training set) even when predicting encodings for problem classes not present in the training set. The custom setup is also shown to greatly outperform the sophisticated algorithm selection tool AutoFolio [4]. In that work the choice was made once for each relevant constraint type. In this work we also use specialised features, but we make the selection for each individual PB or LI constraint.

1.1 Motivation

The ML setup discussed in [9] was shown to be valuable in learning to select good SAT encodings for PB and LI constraints, especially when using features of the specific constraint types. However, choosing one encoding for all constraints of a given type in a problem instance potentially leads to two issues. Firstly, any given problem instance might contain many constraints of the same type but with quite different features. A single encoding selection may not be the best for all the constraints of that type. Secondly, the features of individual constraints are combined in [9] in order to produce a feature vector per instance. This means valuable information may be lost by aggregation.

Two questions naturally arise. Is it practical to train an ML system to predict SAT encodings for each individual constraint of a given type? And if so, how does the performance compare to making one choice per constraint type in a CSP instance? In this work we address these questions, describing and evaluating an ML-based approach to learning to predict encodings at the individual constraint level. We refer to this system as IndiCon. For ease of comparison, we refer to the approach set out in [9] as LeaSE-PI (for "Learning to Select Encodings Per Instance").

1.2 Contributions

In summary, our contributions are as follows:

- We address the problem of selecting SAT encodings for individual PB and LI constraints in instances of CSPs from *unseen* problem classes.
- We present and discuss how to obtain useful training data for individual constraints.
- We adapt and extend the *lipb* features from [9] for PB and LI constraints, in order to describe individual constraints.
- We evaluate empirically a number of alternative setups for our approach.

The focus here is not primarily on performance compared to earlier work, but rather on scientific and methodological contributions:

- It is natural to ask whether setting constraint encodings individually is more effective than setting the encoding for all constraints of one type. Essentially we are investigating this question by building the IndiCon system.
- IndiCon scales better than the earlier LeaSE-PI approach, as discussed below.
- IndiCon produces simpler ML models and that is beneficial for explainability of decisions.

Note: this work is from a PhD thesis [7], which contains many further details.

1.3 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

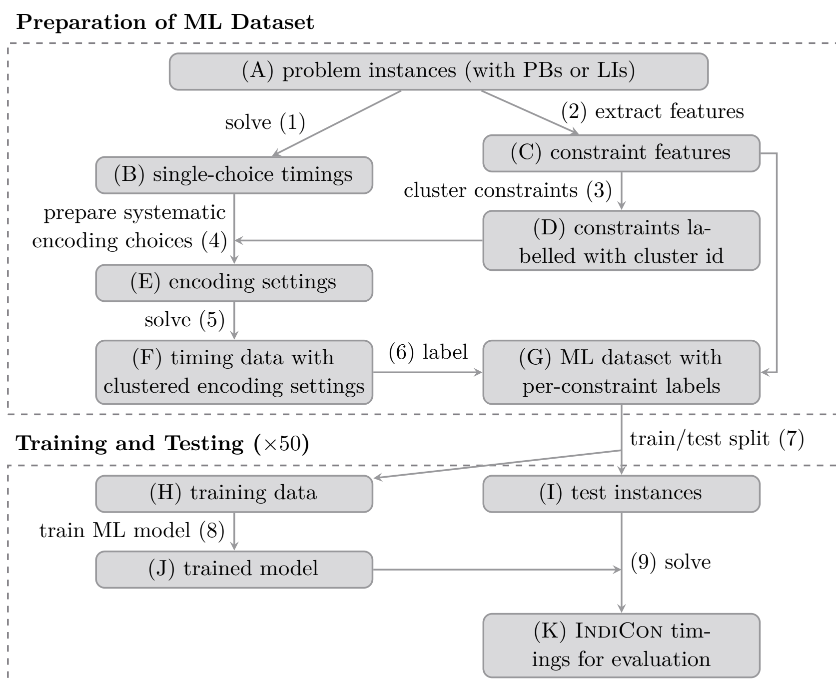


Figure 1 The steps involved in IndiCon. The boxes represent data; the arrows show processes.

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. Pseudo-Boolean (PB) and Linear Integer (LI) constraints are in 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 Boolean or integer decision variables for PB and LI constraints respectively. An *at-most-one* (AMO) constraint over a set of Boolean decision variables requires that zero or one of them are set to true.

2 Method

We begin with a summary of the steps involved in IndiCon, summarised in Figure 1.

1. We start with a corpus of problems (A). We solve the instances initially with each single encoding choice per constraint type and record the timings (B). These allows us to identify a good default encoding choice for the instance.
2. We extract features of each individual PB or LI constraint in the problem instances (C).
3. We use a clustering algorithm to group all the constraints across all instances into clusters with similar features (D).
4. We prepare a number of encoding settings (E) for each instance so that we can systematically try different encodings for constraints by cluster.

5. Each problem is solved using SavileRow with the per-cluster encoding settings (E) and we record the runtimes (F).
6. The timing results (F) allow us to generate a training set (G) with the best encoding label for each constraint.

We are now in a position to configure and train an ML model on the features and labels obtained above. For the sake of robustness, the entire process of splitting the corpus, training and testing is repeated 50 times for each setup.

7. The corpus of problems (A) is split into training and testing instances (H,I), keeping instances from the same problem class together, i.e. only in either the training or test set.
8. An ML model is trained (J) to predict per-constraint encodings.
9. To evaluate performance, we solve the instances in our test set (I), consulting the ML model (J) to decide which encoding to use for each individual constraint, and recording the time taken (K) in order to analyse the performance.

In the rest of this section we briefly present the details of the steps introduced above. Further details, results and discussion can be found in [7].

2.1 Problem Corpus

The problems used are largely the same as in LeaSE-PI [9, 8] (with the addition of more nurse scheduling problems based on instances in NSPLIB [10]), providing a variety of problem categories. The corpus consists of 50 constraint models with up to 50 instances each. There are 551 instances featuring PBs and 347 with LIs. Constraint models include problem classes such as nurse scheduling, car sequencing, knapsack, n-queens, balanced incomplete block design (BIBD), quasigroups, multi-mode resource-constrained project scheduling (MRCPSP), and optimum portfolio design.

2.2 Menu of PB and LI Encodings

We use the same set of 9 encodings for PB and LI as in [9]: the 8 PB(AMO) encodings described and analysed in [2] (GGPW, GGT, GGTd, GLPW, GMTO, GSWC, MDD, RGGT) as well as the non-AMO-aware *Tree* encoding [9]. A PB(AMO) constraint is a PB constraint where the decision variables are partitioned into subsets, each subject to an AMO constraint. The AMO partition is automatically detected [1] for existing PB constraints. LI constraints (if not encoded with *Tree*) are reformulated into PB(AMO)s where each integer variable is represented by a set of Boolean variables with an AMO constraint. An equality (either PB or LI) can be either encoded with *Tree* or broken down into two inequalities and encoded with a PB(AMO) encoding.

2.3 Feature Extraction

We adapt the feature extraction to consider the same aspects of PB/LI constraints as in the *lipb* featureset [9], but without aggregating over all constraints in the instance. In addition, we extract the following:

- `is_equality` records whether the constraint is an equality (rather than $\leq$).
- `amog_maxw_med` is the median maximum weight across AMO groups and gives more information about the distribution of maximum weights in the AMO groups when coupled with the existing mean measure `amog_maxw_mn`.

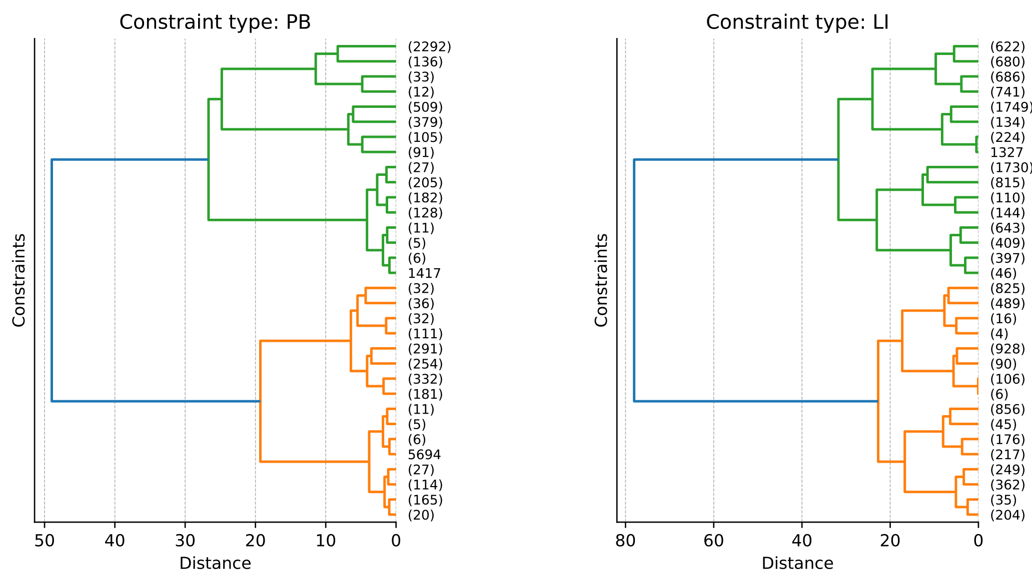


Figure 2 Dendrograms showing agglomerative clustering by constraint features. The x-axis shows the Euclidean distance between clusters. On the y-axis labels indicate the number of data points in a branch (in brackets); labels without brackets identify single-constraint clusters.

- `amog_maxw_mn2k` is the ratio of the mean of maximum coefficients in the AMO groups to the upper limit k and could be an indication of how difficult the constraint will be to satisfy, with a higher value meaning a tighter constraint.
- `amog_maxw_sum` (the sum of the maximum coefficients) tells us the size that the "left-hand side" of the comparison could potentially reach.

2.4 Obtaining Training Data

The requirement for our training data is a feature vector per constraint (for the constraint type in question) as well as a target label to learn. We initially solve each instance in the corpus using each of the encodings available to establish a baseline default encoding for each problem instance. To obtain the target label, we adopt two approaches.

Inheriting from Host Instance In this approach we use the baseline encoding of the instance (as described above) as the target label for all the constraints in the instance. This is an easy way to generate a training set, but it relies heavily on the assumption that constraints of one type will have similar characteristics within one problem instance, which seems in opposition to the motivation behind IndiCon. Nevertheless, it turns out to be a useful method overall.

Clustering Across the Corpus We cluster the individual constraints into groups across all the instances in the corpus. Many unsupervised learning algorithms exist which can generate clusters from data points. We use agglomerative clustering (as implemented in [5]) because it allows us to choose the number of clusters according to the data, rather than having to specify it arbitrarily. Each feature vector of a constraint begins by being its own cluster. As we increase the allowed distance between points in a cluster, clusters merge. This is illustrated in Figure 2. In the left dendrogram (for PBs), we see for example that as the

inter-cluster distance passes 12, 6 clusters become 5. The data remains split into 5 clusters until the distance is 19. Then, the clusters keep joining quite quickly until we reach a distance of 27 and we're left with 2 clusters. For our purpose we observe that 2 clusters and 5 clusters cover the largest ranges of distances. Of these we choose 5 clusters as this allows us to try more configurations for our encodings while still being practical to implement. Using similar reasoning for the LI constraints, we select 6 clusters.

Systematically timing combinations of encodings Once each individual constraint is associated to a cluster, we need to try different combinations of encodings by cluster in order to obtain runtimes and thereby determine which encoding to learn for the constraints in each cluster. Recall that we have 9 candidate encodings. Consider an instance with 5 clusters of constraints – the comprehensive approach would be to try every possible combination; however, that would amount to 9^5 combinations. A compromise is to choose the baseline encoding for all constraints and then change the encodings of one cluster of constraints in turn. This way the example above would require $(1 + 8 \times 5) = 41$ combinations. Solving the instance 41 times remains practical to implement and the number of runs needed scales linearly with the number of encoding choices for a fixed number of clusters.

2.5 Training and Testing

We split our corpus into training and test sets randomly, ensuring that instances of a problem class are either in the training or test set, never in both. This means when we are attempting the challenging task of making predictions for unseen problem classes. The train/test split is approximately 80% : 20% (approximate because of the different numbers of instances available for each problem class). We carry out 50 splits with different random seeds.

Similar to [9] we train classifiers to select between pairs of encodings; the final prediction is the result of voting from the trained classifier models. We separately train random forests, gradient boosted trees and simple decision tree models. In early experiments we also tried k-nearest neighbours and simple neural networks, as well as an ensemble of all classifiers mentioned above, but the performance was worse than with the classifiers we present here.

2.6 Experimental Setup

All experiments were carried out on a high-performance cluster with Intel Xeon 6138 20-core 2.0 GHz processors; the memory limit per job was set to 6GB. SavileRow was run with AMO detection switched on, a SAT clause limit of 10 million, and a timeout of 1 hour. Kissat (`sc2021-sweep`) was used, with its own 1-hour timeout. Each solving run is repeated with 5 different seeds; the median runtime is then calculated and a 10-fold penalty is applied for any total runtime over 1 hour to give PAR10 results.

3 Results and Discussion

The corpus contains problem classes with PBs, LIs, or both. We apply IndiCon separately to these two constraint types, using the problems containing the relevant constraint type.

3.1 Selecting Encodings for One Constraint Type

To evaluate the performance of IndiCon, we record the PAR10 running time for the 50 test sets as a multiple of the virtual best time achievable by using a single encoding. It is

Table 1 IndiCon performance for the best 3 setups, ordered from best to worst performing. Each setup is tested over 50 train/test splits. The performance is measured using PAR10 and shown as a multiple of the Virtual Best* (* single-choice) time. For reference, Single Best time is also shown.

IndiCon for PB

Setup		Runtime
Clusters	Classifier	PAR10 × VB*
1	RF	5.57
1	DT	5.69
5	GB	8.10
Single Best		*11.58*

IndiCon for LI

Setup		Runtime
Clusters	Classifier	PAR10 × VB*
Single Best		*4.53*
6	RF	6.44
1	RF	6.70
6	GB	11.12

prohibitive to calculate a true virtual best by running every single combination of encodings for every constraint in any sizeable instance. Our reference is called *VB** to emphasise that this is a single-choice virtual best. We also show the single best (SB) result, which is the result of always choosing the one encoding which performed best on the training set.

The results are shown in Table 1. For this corpus IndiCon seems to work better for PBs than for LIs, with the best setup achieving 5.57 times the VB* time, compared to 11.58 times for the single best (SB) time. In the LI setting, IndiCon does not even match the SB performance of 4.53 on this corpus, coming in at 6.44 times VB*. This may be explained to an extent by the fact that in [9] the authors found that the choice of PB encoding could make a much bigger difference to solving performance, whereas for LIs it tended to be the case that there was one encoding (GGPW) which usually outperformed the others.

In terms of the classifiers used, it is interesting that a simple decision tree classifier employed in a pairwise voting setup is performing almost as well as random forests – this would open up the way to more explainability for the encoding choice made.

The best setups for PB come from the non-clustered training data, i.e. where the target label for training was simply the encoding which worked best for the host instance of every constraint. In the LI setting, the setup using 6 clusters slightly outperforms the setup based on the simpler labelling source.

3.2 Comparison with Per-Instance Selection in LeaSE-PI

We carry out a second experiment in order to compare the performance of IndiCon with LeaSE-PI. We consider the 250 instances which contain both PB and LI constraints and appear in at least one of the LeaSE-PI and one of the IndiCon test sets. For each instance we randomly sample with replacement 100 results from LeaSE-PI and 100 results of running IndiCon to select both PB and LI encodings. Each solving run is done 5 times and the median time is recorded to account for randomness in SAT solving.

The results are shown in Figure 3. The left plot shows how many instances were solved as we increase the CPU time. We see that IndiCon is competitive with LeaSE-PI and does better for some of the harder instances which take around 3000 seconds to solve. This slight edge is confirmed by the mean PAR10 solving time across the 25000 "contests": for LeaSE-PI the mean is 1161 seconds with 689 timeouts, and for IndiCon the mean is 1145 seconds with 668 timeouts. From the scatter plot on the right of Figure 3 we see a fairly consistent performance between the two selectors, without any extreme differences as there are no crosses in the top left or bottom right corners. The curve at the bottom left indicates that LeaSE-PI is doing better on the easiest instances, whose runtime is under 1 second.

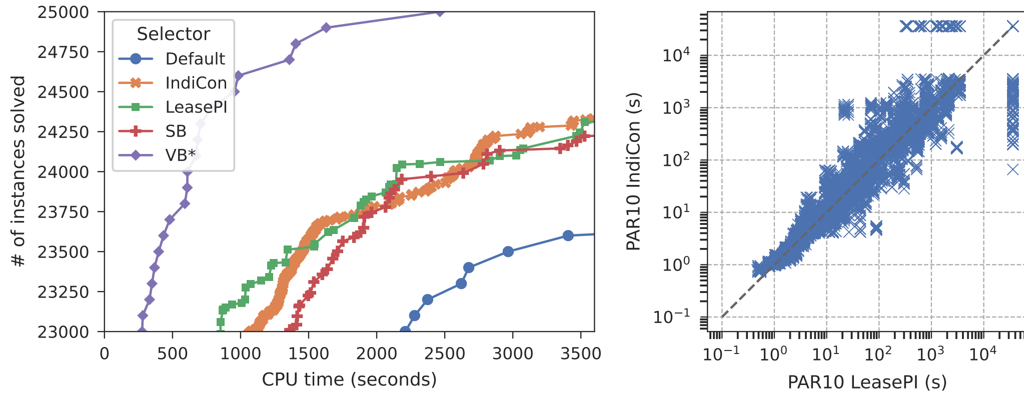


Figure 3 Comparison of IndiCon and LeaSE-PI performance on sample of instances when IndiCon has been used to set both PB and LI constraints. Left: number of instances solved by CPU time up to 1 hour for the single-choice virtual best (VB*), single best (SB), default encoding (Def), best LeaSE-PI setup and IndiCon. Right: PAR10 times for the best LeaSE-PI versus IndiCon.

The overhead of retrieving a separate encoding choice for each constraint from the ML classifiers becomes less significant as the overall solving times increase for harder problems.

The LeaSE-PI paper [9] has a section (Analysis of the configuration space) which shows that the best choice for LI was much more dominated by one good choice, whereas for PB constraints the best choice of encoding was much more varied. This is not fully explained, but it could be that in the corpus we used the sum (LI) constraints were encoding larger integer values, whereas the coefficients in the PBs were smaller. The mean value for median coefficient in a PB is 1.03, compared to 142 for LI constraints. In IndiCon we once again find that selecting encodings for PB constraints affects the performance more than for LI constraints, as discussed in Section 3.1.

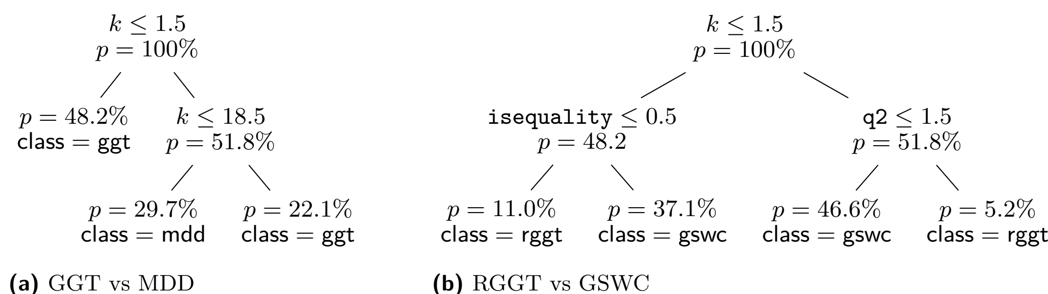
A final observation is that in LeaSE-PI the ML setup is able to take into account both types of constraints across the entire instance, so to some extent it could learn combinations of constraint choices based on how decision variables are shared between the different constraints. Here, we make the choice in isolation for each constraint type but are still able to match the performance of LeaSE-PI. We did run trials which included whole-instance features in the IndiCon training data, but performance actually suffered.

3.3 Explaining Decisions using Decision Trees

We noted that decision trees in a pairwise arrangement perform almost as well as random forests for selecting PB encodings. To illustrate how this selection is made, we show in Figure 4 a small sample of the decision trees resulting from training to select the PB encoding. In the first example (a), we see that GGT is chosen when the upper limit k is either 1 or above 18, whereas MDD is preferred for intermediate k values. In example (b), GSWC is preferred to RGGT either when $k = 1$ and the comparison is equality, or where k is larger, but the median coefficient is below 2.

4 Related Work

LeaSE-PI selects SAT encodings per constraint type using ML; in [9] the authors compare LeaSE-PI's performance with AutoFolio [4] which is a sophisticated (albeit general)



(a) GGT vs MDD **(b)** RGGT vs GSWC

Figure 4 Sample decision trees from pairwise training to select PB encodings. The p values are the proportion of training samples; the symbols on the left side of operators are constraint features, e.g. k is the upper limit of the PB or LI constraint, $q2$ is the median coefficient, and isequality is 1 for an equality constraint and 0 otherwise.

algorithm selection tool. LeaSE-PI significantly outperforms AutoFolio on the specific task in question. In this paper we show that IndiCon performs slightly better even than LeaSE-PI. More generally, the use of ML to select SAT encodings for integer variables is addressed by MeSAT [6]. Proteus [3] also makes this kind of choice based on CSP instance features, having first chosen whether to use a SAT solver at all (as opposed to a constraint solver); it also goes on to predict which SAT solver to use.

5 Conclusion

We have presented IndiCon, an ML system for selecting SAT encodings of individual constraints in a CP model. To our knowledge, IndiCon is unique in choosing an encoding for each constraint separately. We have shown that the performance of IndiCon for selecting both PB and LI constraint encodings is marginally better than the existing state of the art. The key benefits of IndiCon compared to the prior work are *scaling* and *simplicity* (leading to explainability). It treats each constraint type as a separate ML problem, and as a consequence it scales linearly in the number of constraint types (unlike LeaSE-PI [9], the best version of which scales exponentially in the number of constraint types). IndiCon typically learns simpler models than LeaSE-PI, which benefits explainability of the system. Even very simple ML models such as decision trees can provide competitive results in this context.

References

1 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.

2 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.

3 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.

4 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.

5 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.

6 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.

7 Felix Ulrich-Oltean. *Learning SAT Encodings for Constraint Satisfaction Problems*. PhD thesis, University of York, September 2023.

8 Felix Ulrich-Oltean. Learning to Select SAT Encodings - Data and Code for Experiments. https://github.com/felixvuo/lease-data, January 2023.

9 Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Learning to select SAT encodings for pseudo-Boolean and linear integer constraints. *Constraints*, November 2023. doi:10.1007/s10601-023-09364-1.

10 Mario Vanhoucke and Broos Maenhout. NSPLib – A Nurse Scheduling Problem Library: A tool to evaluate (meta-)heuristic procedures. page 11.