Selecting SAT Encodings for Pseudo-Boolean and Linear Constraints: Preliminary Results

Felix Ulrich-Oltean$^{1\,[0000-0001-5162-5826]}$,
Peter Nightingale$^{1\,[0000-0002-5052-8634]}$, and
James Alfred Walker$^{1\,[0000-0003-2174-7173]}$

Department of Computer Science, University of York, York, United Kingdom
{fvuo500,peter.nightingale,james.walker}@york.ac.uk

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 integer constraints. We briefly discuss the relative importance of instance features to the task of selecting the best encodings.

Keywords: SAT encodings · machine learning · global constraints

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 $\diamond \in \{<, \leq, =, \neq, \geq, >\}$, $q_1 \dots q_n$ 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* 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.

We select from a set of state-of-the-art encodings, including all four encodings of Bofill et al [8,9] which are extensions of the Generalized Totalizer [15], Binary

Decision Diagram [2], Global Polynomial Watchdog [6], and Sequential Weight Counter [13]. 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.

The context for this work is Savile Row [19], 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 to SAT, for each constraint type in the language, Savile Row either decomposes it (e.g. allDifferent) or applies the chosen SAT encoding (or the default if no choice is made).

We use a supervised machine learning approach, trained with a corpus of 50 problem classes with 622 instances. 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.

1.1 Preliminaries

A *constraint satisfaction problem* (CSP) is defined as a set of variables X, a function that maps each variable to its domain, $D : X \rightarrow 2^{\mathbb{Z}}$ 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 sequence 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 enforces GAC if this domain reduction in the source CSP is reflected in the encoding's SAT variables and clauses.

2 Learning to Choose SAT Encodings

First we describe the palette of encodings for linear integer and pseudo-Boolean constraints, then our approach to selecting encodings using problem 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. We use 5 encodings and each can be applied to either pseudo-Boolean or linear integer constraints, giving 25 configurations in total. The first 4 are encodings of PB(AMO) constraints [8,9], 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]. For the 4 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 (i.e. must have a corresponding SAT variable). All PB(AMO) encodings require a direct encoding of integer CSP variables, and when an integer variable (with d values) appears in a linear integer constraint it is replaced with an AMO group of $d-1$ terms representing each value except the smallest (which is cancelled out). Full details of the conversion of integer terms and normalisation are given elsewhere [5]. 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].

The Multi-valued Decision Diagram (*MDD*) encoding [8] (a generalisation of the BDD encoding for PB constraints [2]) 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. The Generalized Global Polynomial Watchdog (*GGPW*) encoding [9] (generalising the GPW [6]) is based on bit arithmetic and is polynomial in size, however unit propagation on GGPW does not achieve GAC on the original constraint. The Generalized Generalized Totalizer (*GGT*) [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 is able to compress equivalent states at its internal nodes. It generalises the Generalized Totalizer [15]. The Generalized Sequential Weight Counter (*GSWC*) [9] (based on the Sequential Weight Counter [13]) 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 [9] but not when c contains integer terms or is an equality or disequality. GGPW does not have this property.

Finally, the *Tree* encoding is related to GGT however it is not a PB(AMO) encoding. 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 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[1]

[1] Savile Row generates the *direct* or *order* encoding for variables as required [18].

and is also used for internal nodes. Each internal node is connected to its two children using the order encoding of linear constraints [22]. 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]. Log encodings such as the one used by PicatSAT [23] may be more effective in some cases. For our experiments we use an extended version of Savile Row 1.9.0 [18]. All constraints other than linear integer and PBs use the default encoding as described in the Savile Row manual.

2.2 Instance Features

We use the 95 features extracted by the `fzn2feat` tool produced by Amadini et al. [4] as a starting point. We use `fzn2feat` directly in our experiment, but we have also re-implemented the set of 95 features as closely as possible within Savile Row, applied to the model directly before encoding to SAT. In addition, we propose 27 new features that are specific to linear integer or PB constraints (and we generate these for both classes of constraint, giving 54 new features in total). The new features are:

- The number of (linear or PB) constraints (1 feature)
- For the number of terms in each constraint: min, max, mean, median, inter-quartile range, non-parametric skew, overall sum (7)
- The overall sum of all the coefficients (1)
- For the minimum coefficient in each constraint: min, mean (2)
- For the maximum coefficient in each constraint: max, mean (2)
- For the median coefficients: median, non-parametric skew, Shannon's entropy (3)
- For the sums of coefficients: non-parametric skew, inter-quartile range (2)
- For the inter-quartile range of coefficients: median, non-parametric skew (2)
- For the coefficients' quartile skewness $\frac{(Q_3-Q_2)-(Q_2-Q_1)}{Q_3-Q_1}$: mean, min, max (3)
- For the number of distinct coefficient values: mean, max (2)
- For the ratio of distinct coefficient values to number of coefficients: mean, max (2)

2.3 Problem Corpus

We used all problems from a recent paper with a corpus of 65 constraint models and a total of 757 instances [12]. This collection has a very skewed distribution of instances per problem class, ranging from just 1 to 100. We addressed this by adding some problem instances to existing classes and by limiting the number of instances per class to 50. We also added two problems (Hamiltonian Cycle and Balanced Academic Curriculum) from recent XCSP3 competitions [1]. We dropped instances from the corpus if they contained no linear integer or PB constraints.

2.4 Training and Prediction

We have evaluated several classifier models from the `scikit-learn` library [20] and have found that `RandomForestClassifier` performs best for our purposes. The 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 use randomised search with 5-fold cross-validation to set the hyper-parameters (criterion, number of estimators, maximum tree depth and maximum features).

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 mis-classification can be very expensive. To address this issue, we follow a similar approach to the *pairwise classification* used in AutoFolio [17]: we train a random forest model for each pair 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 encodings from the features and leads to better prediction performance than using a single random forest classifier.

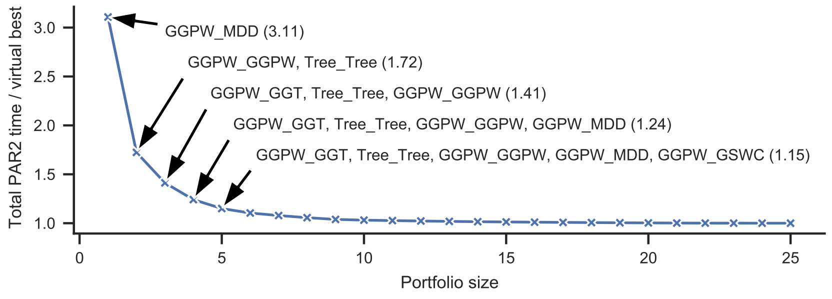

Fig. 1. The virtual best PAR2 run-time on our corpus for all portfolio sizes as a multiple of the overall virtual best; the resulting portfolios (of *li_pb* configurations) are shown for sizes 1 to 5.

To facilitate this pairwise training and prediction approach, we reduce our portfolio of encoding choices to 5, thus needing to train 10 models (rather than 300 if we had used all 25 choices). We seek to retain the performance complementarity described in [16] from a much reduced portfolio size. We build the portfolio using a greedy approach, beginning with a single encoding configuration in the portfolio and then successively adding in whichever remaining configuration would lower the virtual best PAR2 time (PAR2 is defined in Section 3.2) by the biggest margin. We do this until we have our 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 portfolios. 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 same measure across all 25 configurations.

3 Empirical Investigation

3.1 Solving Problem Instances and Extracting Features

We ran Savile Row on each instance in the corpus with each of the 25 encoding configurations. The CNF clause limit was set to 5 million and the Savile Row time-out to 1 hour. We switched on automatic detection of At-Most-One constraints [5]. We used the Kissat solver 1.0.3 [7]. Kissat was the winner of the main track of the SAT Competition 2020. Default settings were used for Kissat, and it was run with a separate time limit of 1 hour. The experiment was run on the Viking research cluster with Intel Xeon 6138 20-core 2.0 GHz processors; we set the memory limit for each job to 8 GB. We took the median of 5 runs (with 5 distinct random seeds) for each configuration to average out stochastic behaviour of the solver. To extract the features we ran each problem instance once with the Savile Row feature extractor and once to generate standard FlatZinc (using the `-flatzinc` flag) followed by `fzn2feat` [4].

3.2 Labelled Datasets and the Training/Test Split

The first step is to process our total (Savile Row + Kissat) runtimes and filter the corpus. We mark a result as timed out if the total runtime exceeds 1 hour. We use PAR2 times, i.e. assigning 2 $\times$ time limit to any result which takes longer than our time-out limit. 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, 622 instances of 50 problem classes remain in the corpus; the number of instances for each problem class is shown in Figure 2.

We build four datasets of features: *f2f* for the `fzn2feat` features, *f2fsr* for the equivalent features extracted directly from Savile Row's internal model prior to SAT encoding, *lipb* for our new features of linear integer and pseudo-Boolean constraints alone, and *combi* for *f2fsr* and *lipb* combined. Each row in each of these datasets represents one instance, and includes the names of the model and parameter files as well as the instance features. To enable training of one of the pairwise models (as described in Section 2.4), we label each row with the better configuration (of the two under consideration).

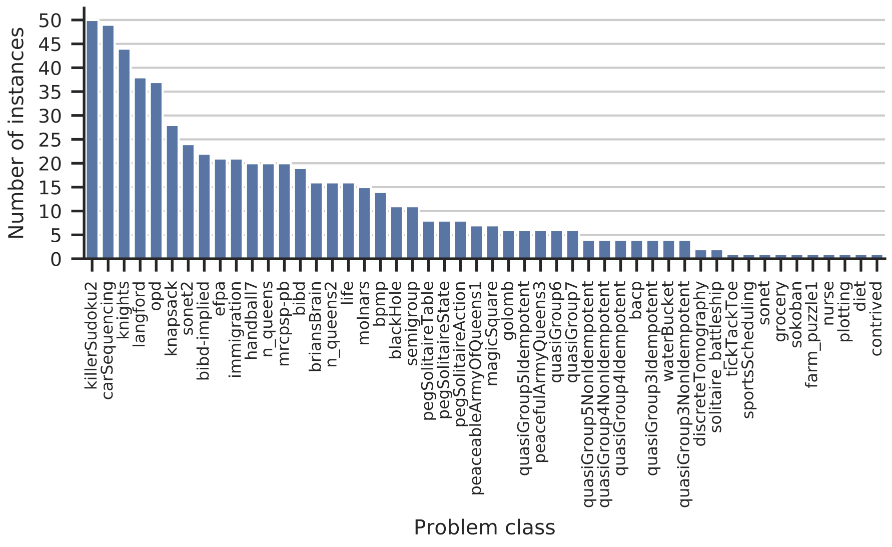

Fig. 2. The number of instances for the different problem classes (models) in the corpus after dropping instances that were fully solved by Savile Row or timed out for all encoding configurations.

We run the *split, train, predict* process 10 times on each of the four datasets, using seeds $\{1 \dots 10\}$ to co-ordinate the splits so that we compare the prediction power of the different feature sets using the same training and test sets. For each *split, train, predict* process, we split each dataset into an 80% training set and a 20% test set using simple uniform random sampling.

3.3 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 [16]: the *Virtual Best Configuration (VBC)* time is the total time taken to solve the instances in the 10 test sets if we always made the best choice from our portfolio of configurations; the *Single Best Configuration (SBC)* time is how long it would take using the one configuration which performs fastest on our entire corpus – in our case this is the $\langle li \rightarrow GGPW, pb \rightarrow MDD\rangle$ configuration. In addition we refer to: the time taken using Savile Row's *default (Def)* configuration, which is the *Tree* encoding for both linear integer and pseudo-Booleans, and finally the *Virtual Worst Configuration (VWC)* to indicate the overall variation in performance of the encodings in the portfolio. We can then report the solving time for the predicted encodings, for each of the four feature sets. We also record the time taken to extract the features; the predicted time with this added time cost is

Table 1. Total PAR2 times over the 10 test sets as a multiple of the virtual best configuration time. We show the times for the virtual best (VBC), virtual worst (VWC), single best (SBC), and default (Def) configurations, followed by the timings using predictions made on our four feature sets, without and with feature extraction (FE) time. The best time (including FE) is shown in **bold**.

Benchmarks				Predicted				Predicted + FE Time			
VBC	VWC	SBC	Def	f2f	f2fsr	lipb	combi	f2f	f2fsr	lipb	combi
1.00	6.66	2.72	3.78	1.76	1.89	1.67	1.60	1.80	1.91	1.69	**1.62**

reported using the suffix $+FE$. In all cases we calculate the total time for each one of the 10 test sets, then sum the 10 test sets to obtain the overall total time.

3.4 Results and Discussion

We found that the machine learning predictors work well, and that their performance approached that of the virtual best configuration. Table 1 shows summary statistics for each feature set. The mean PAR2 time to solve the test sets is reported, scaled so that the virtual best configuration (VBC) would be 1 (e.g. a value of 2 in Table 1 means double the time of the VBC). The generic CSP feature sets *f2f* and *f2fsr* improve on the SBC, but the new features are significantly better and the best result was obtained by *combi*, combining *f2fsr* and *lipb*. 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" [16]. In these terms, *f2f* has closed 53% of the VBC-SBC gap, and *combi* has closed 64% of the gap.

Figure 3 shows the performance profile of each predictor, the VBC, the default setting of Savile Row (Def), and the SBC (on the 10 test sets). Instances are sorted by VBC solving time to place the most difficult instances first on the x-axis. The default configuration of *Tree_Tree* is clearly performing poorly for many of the most difficult instances, despite being the configuration that is most frequently the best. The SBC (*GGPW_MDD*) performs much better on the most difficult instances. It may be a factor that *Tree* is the only encoding that is not able to take advantage of automatic AMO detection (which reduces the size of encodings without changing their propagation properties and is known to help substantially with difficult instances of some classes [5]).

Figure 4 shows (for each predictor and the VBC) the frequency of selecting each of the 5 configurations in the portfolio. It is notable that all predictors choose *Tree_Tree* more frequently than the VBC does. They underpredict *GGPW_GGT* and *GGPW_GSWC*. Correcting these imbalances in some way could lead to better performance.

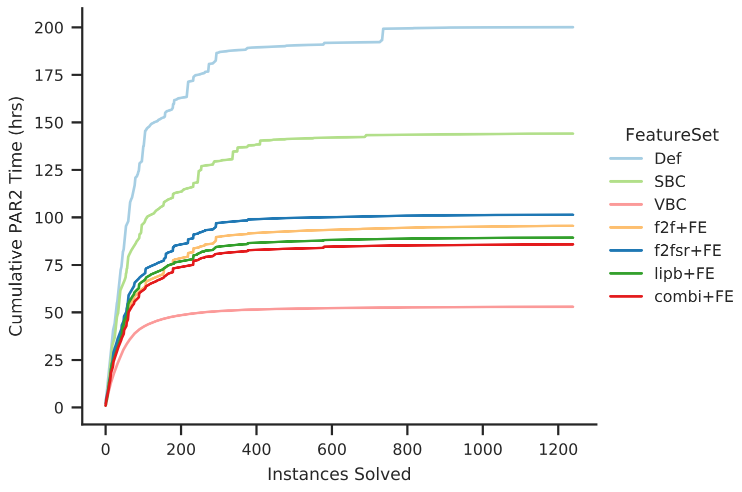

Fig. 3. Cumulative PAR2 time over the 10 test sets, with instances sorted (by VBC solving time) to place the most difficult instances first on the x-axis. We show the times for the virtual best (VBC), single best (SBC) and default (Def) configurations, and the timings using predictions made on our four feature sets with feature extraction (FE) time.

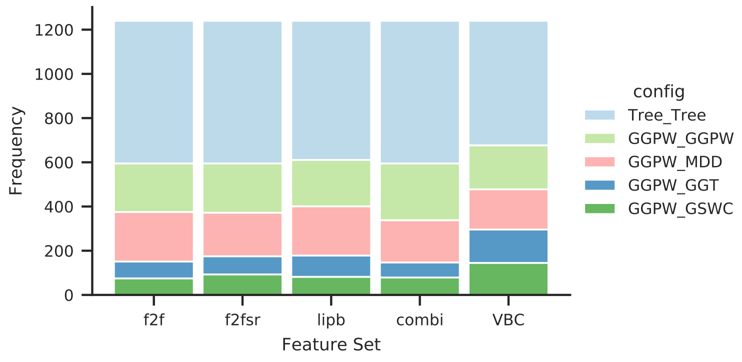

Fig. 4. Frequency of each configuration (*li_pb*) predicted for the 10 test sets when using each feature set. We also show the virtual best configuration for comparison.

3.5 Feature Importance

The random forest classifier provided by `scikit-learn` [20] can report the relative importance of the features in the training phase. We extract the Gini (impurity-based) feature importance from every pairwise training phase (5 encoding configurations lead to 10 pairs) across all 10 runs. The most important features are shown in Figure 5. There is huge variability in the relative importance of any given feature across the different train/test splits and across the different pairwise match-ups. Nevertheless we can make some observations.

The relative overall importance of features decreases very slowly, meaning that there are no outstanding features which discriminate far better than others. We show only 25 features for readability, but this trend continues for the other features too.

We suspect that the classifier is, to a large extent, recognising problem classes rather than picking out traits of LI/PB constraints. For instance notice that in the *combi* set, none of the *lipb* features make the top 25. The highest ranked *lipb* feature (*pbs_x_sum* which gives the total number of terms in all PBs in an instance) is in 28th place. Recall the distribution of instances across problem classes shown in Figure 2: there are some problems with very few instances (10 have only 1 instance). This means that many problem classes will only have instances exclusively in either the train or the test, making the classification harder.

Considering our new *lipb* features we are encouraged that using these alone was enough to outperform the classification performance of the generic instance features (*f2f* and *f2fsr*), although, as seen previously in Figure 3, the best performance was achieved by combining the LI/PB and generic features.

We note that of our new *lipb* features, the top few seem to make markedly more difference than the rest – this may simply reflect that there are fewer features in this set (54 compared to 95 or 149). Of the 5 most discriminating features in this set 4 are related to the size of the pseudo-Boolean constraints: *pbs_x_sum* is the total number of terms across all PBs, *pbs_w_sum* the sum of all coefficients in PBs, *pbs_w_avg* the mean number of terms in PBs and *pbs_count* gives the number of PBs.

One final observation we make is that choice of SAT encoding for pseudo-Boolean constraints appears more important, or at least harder to make, than for linear integer constraints. When constructing our portfolio of 5 encodings (see Figure 1), the *GGPW* encoding was selected for LI constraints in 4 of the 5 configurations, whereas all 5 options were present in the PB slot. This observation is also backed up by the higher ranking of *pbs_* features as opposed to *sums_* in the *lipb* set.

4 Related Work

MeSAT [21] and Proteus [14] both select SAT encodings using machine learning. MeSAT has two encodings of linear integer constraints: the order encoding [22];

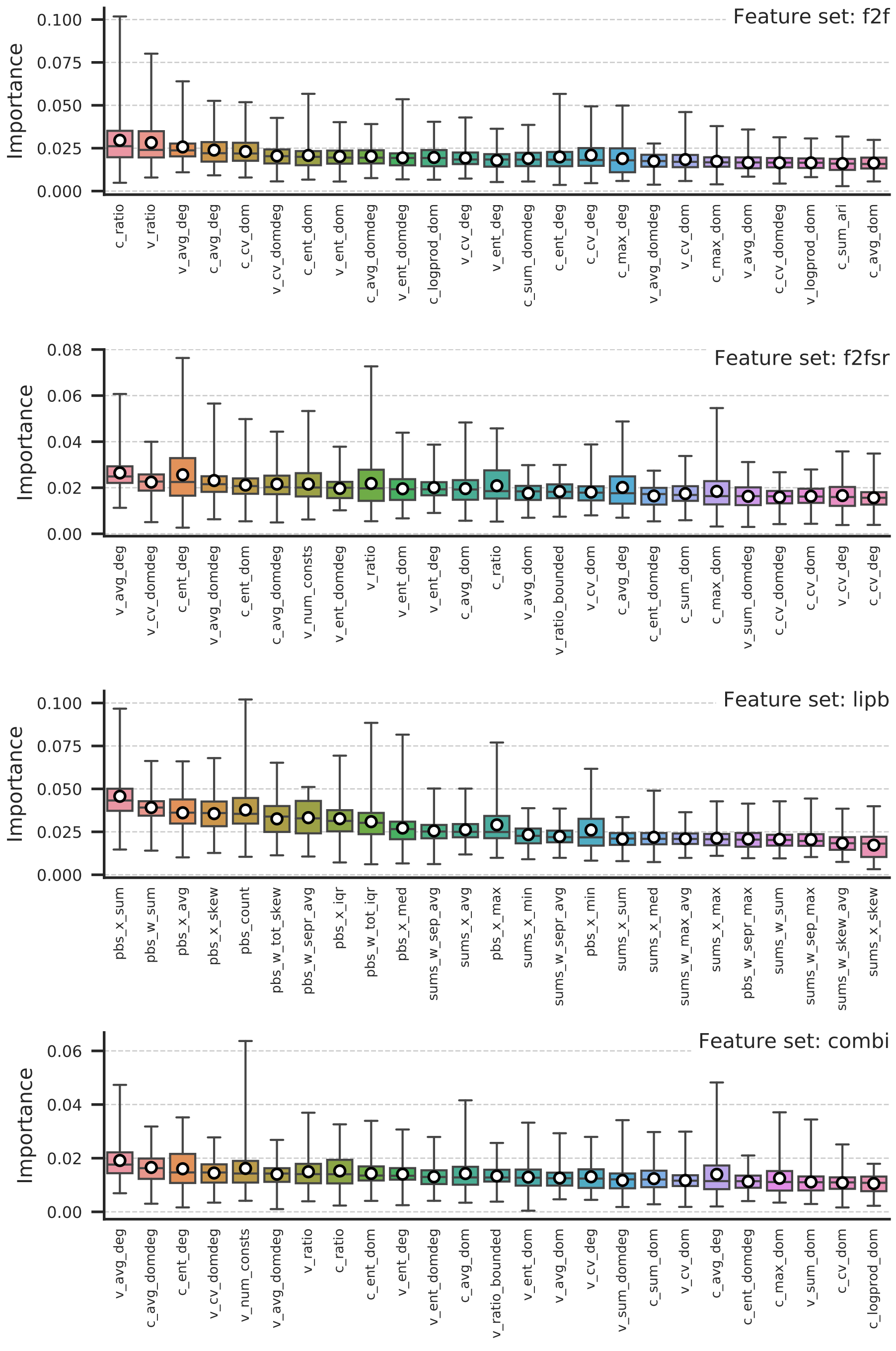

Fig. 5. Gini importance of features for classification. Each box plot summarises 100 training epochs arising from 10 pairwise settings and 10 trials. The mean feature importance is shown by a small circle. Only the top 25 features are shown here (by median importance). The *lipb* features were introduced in this paper in Section 2.2. The *f2f* features from `fzn2feat` are described at `https://github.com/CP-Unibo/mzn2feat/`.

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 choice of 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 linear integer constraints is not specified [14]. 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 encodings by using machine learning to select encoding configurations based on instance features. 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 observe that there is more scope for performance gains in selecting encodings for pseudo-Boolean than for linear integer constraints.

We hope to build on these promising 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.

Acknowledgements

Felix Ulrich-Oltean is supported by grant EP/R513386/1 from the UK Engineering and Physical Sciences Research Council. 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 computational support from the University of York High Performance Computing service, Viking and the Research Computing team.

References

1. 2019 XCSP3 Competition (2019), `http://www.cril.univ-artois.fr/XCSP19/`
2. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A New Look at BDDs for Pseudo-Boolean Constraints. Journal of Artificial Intelligence Research **45**, 443–480 (Nov 2012). https://doi.org/10.1613/jair.3653

3. Abío, I., Mayer-Eichberger, V., Stuckey, P.J.: Encoding linear constraints with implication chains to CNF. In: International Conference on Principles and Practice of Constraint Programming. pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-23219-5_1
4. Amadini, R., Gabbrielli, M., Mauro, J.: An enhanced features extractor for a portfolio of constraint solvers. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1357–1359. SAC '14, Association for Computing Machinery, New York, NY, USA (Mar 2014). https://doi.org/10.1145/2554850.2555114
5. Ansótegui, C., Bofill, M., Coll, J., Dang, N., Esteban, J.L., Miguel, I., Nightingale, P., Salamon, A.Z., Suy, J., Villaret, M.: 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. pp. 20–36. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7
6. Bailleux, O., Boufkhad, Y., Roussel, O.: New Encodings of Pseudo-Boolean Constraints into CNF. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009. pp. 181–194. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_19
7. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series b, vol. B-2020-1, pp. 51–53. University of Helsinki (2020), `http://fmv.jku.at/papers/BiereFazekasFleuryHeisinger-SAT-Competition-2020-solvers.pdf`
8. Bofill, M., Coll, J., Suy, J., Villaret, M.: 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. pp. 555–562. International Joint Conferences on Artificial Intelligence Organization, Melbourne, Australia (Aug 2017). https://doi.org/10.24963/ijcai.2017/78
9. Bofill, M., Coll, J., Suy, J., Villaret, M.: SAT encodings of pseudo-boolean constraints with at-most-one relations. In: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 112–128. Springer (2019). https://doi.org/10.1007/978-3-030-19212-9
10. Bofill, M., Coll, J., Suy, J., Villaret, M.: An MDD-based SAT encoding for pseudo-Boolean constraints with at-most-one relations. Artificial Intelligence Review **53**(7), 5157–5188 (2020). https://doi.org/10.1007/s10462-020-09817-6
11. Breiman, L.: Random Forests. Machine Learning **45**(1), 5–32 (Oct 2001). https://doi.org/10.1023/A:1010933404324
12. Davidson, E., Akgün, Ö., Espasa, J., Nightingale, P.: Effective Encodings of Constraint Programming Models to SMT. In: Simonis, H. (ed.) Principles and Practice of Constraint Programming. pp. 143–159. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58475-7_9
13. Hölldobler, S., Manthey, N., Steinke, P.: A Compact Encoding of Pseudo-Boolean Constraints into SAT. In: Glimm, B., Krüger, A. (eds.) KI 2012: Advances in Artificial Intelligence. pp. 107–118. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33347-7_10
14. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A Hierarchical Portfolio of Solvers and Transformations. In: Simonis, H. (ed.) Integration

of AI and OR Techniques in Constraint Programming. pp. 301–317. Lecture Notes in Computer Science, Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07046-9

15. Joshi, S., Martins, R., Manquinho, V.: Generalized Totalizer Encoding for Pseudo-Boolean Constraints. In: Pesant, G. (ed.) Principles and Practice of Constraint Programming. pp. 200–209. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_15
16. Kerschke, P., Hoos, H.H., Neumann, F., Trautmann, H.: Automated Algorithm Selection: Survey and Perspectives. Evolutionary Computation **27**(1), 3–45 (Mar 2019). https://doi.org/10.1162/evco_a_00242
17. Lindauer, M., Hoos, H.H., Hutter, F., Schaub, T.: AutoFolio: An Automatically Configured Algorithm Selector. Journal of Artificial Intelligence Research **53**, 745–778 (Aug 2015). https://doi.org/10.1613/jair.4726
18. Nightingale, P.: Savile Row 1.9.0 Manual, `https://savilerow.cs.st-andrews.ac.uk/index.html`
19. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (Oct 2017). https://doi.org/10.1016/j.artint.2017.07.001
20. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
21. Stojadinović, M., Marić, F.: meSAT: Multiple encodings of CSP to SAT. Constraints **19**(4), 380–403 (Oct 2014). https://doi.org/10.1007/s10601-014-9165-7
22. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints **14**(2), 254–272 (Jun 2009). https://doi.org/10.1007/s10601-008-9061-0
23. Zhou, N.F., Kjellerstrand, H.: Optimizing SAT encodings for arithmetic constraints. In: International Conference on Principles and Practice of Constraint Programming. pp. 671–686. Springer (2017). https://doi.org/10.1007/978-3-319-66158-2_43