Learning SAT Encodings for Constraint Satisfaction Problems

Felix Ulrich-Oltean

PhD

University of York

Computer Science

September 2023

Abstract

Constraint programming addresses many interesting and challenging problems in our world, including recent applications to contexts as diverse as allocating refugee relief funds, short-term mine planning and hardware circuit design.

Users define their problems in high-level modelling languages which include descriptive global constraints. One of the most effective ways to solve constraint satisfaction problems (CSPs) is by translating them into instances of the Boolean Satisfiability Problem (SAT). For some global constraints in CSPs there exist many algorithms which encode the constraint into SAT; choosing an appropriate SAT encoding can alter the ultimate solving time dramatically.

We investigate the problem of selecting the best SAT encoding for pseudo-Boolean and linear integer constraints. Many machine learning techniques are explored, applied and evaluated to aid this selection. The result is a significant improvement in performance compared to the default choice and to the single best choice from a training set. The approach is successful even for previously unseen problem classes and it greatly outperforms a sophisticated general algorithm selection and configuration tool.

This work provides a thorough empirical study and detailed analysis of each stage in the machine learning process as applied to choosing SAT encodings. It does this in three phases: firstly by using generic CSP instance features to select an encoding per constraint type for each instance, then by introducing new features which focus on the constraint types in question, and finally by learning to select encodings for individual constraints.

We find that even generic instance features can produce good predictions, but that the specialised features introduced give more robust performance especially when predicting for unseen problem classes. Training to predict per constraint shows potential and leads to better performance for some problem classes, but per-instance selection is still competitive across the corpus of problems as a whole.

Contents

Author Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, university. All sources are acknowledged as references.

In this thesis dissertation, the use of the pronoun "we" refers to the author and reader collectively.

I have presented parts of the work in this thesis previously at the following venues:

CP2020 Doctoral Programme The short paper *Learning SAT Encodings for Individual Constraints* outlines some initial investigations into whether performance gains might be made from selecting different SAT encodings for pseudo-Boolean constraints. The idea is introduced of ultimately selecting encodings for individual constraints.

CP2021 Doctoral Programme My submission *Learning to Choose SAT Encodings for Pseudo-Boolean and Integer Sum Constraints* describes early versions of the work in Chapters 4 and 5.

ModRef 2021 Workshop The paper *Selecting SAT Encodings for Pseudo-Boolean and Linear Constraints: Preliminary Results* [1] written with my PhD supervisors presents some results from Chapters 4 and 5 without predictions for unknown problem classes. It also begins to explore feature importance.

CP2022 Conference Our paper *Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints* [2] written with my supervisors presents the work in Chapters 4 and 5 concisely, including predicting for unknown problem classes and more about feature importance.

Constraints Journal 2023 Our paper *Learning to Select SAT Encodings for Pseudo-Boolean and Linear Integer Constraints* [3] written with my supervisors extends the work of the previous entry to include more encodings and more analysis.

Acknowledgements

Although a PhD is a very individual pursuit, its success depends on the support, guidance and patience of many parties. I am indebted to so many people and organisations who have contributed to my realising a long-held dream. Because ranking importance is a fool's game, I will cowardly opt to present my thanks in a roughly chronological order.

I am immensely grateful to:

- my wife Emma and our children Elisei, Estera, Eva and Ezechiel who encouraged and supported me through this ambition to dedicate several years to in-depth study and research. I thank them for the sacrifices they made to enable me to pursue this dream.
- Elaine Silson, the head teacher Allerton High where I had taught for 10 years, for giving me time and space to explore the possiblity of pursuing a PhD.
- the UK Engineering and Physical Sciences Research Council who funded my PhD with grant EP/R513386/1.
- Peter Nightingale, my principal supervisor, for his patience, care, attention to detail and relentless encouragement through some difficult patches along the way.
- James Cussens (James I) and James Walker (James II) for complementing Pete's advice and for their encouragement and guidance based on years of previous PhD supervision experience.
- the whole Computer Science department at the University of York, in particular the post-graduate admin team, who made me feel welcome and part of the "team" from day one.
- the CP group at University of St Andrews for their virtual open door, their many collaborations and inviting me into their thriving research community. In particular I want to thank Ándras, Chris, Ian G., Ian M., Joan, Mun See, Nguyen, Özgur and Ruth.

- Jordi Coll, Mateu Villaret and Miquel Bofill at Universitat de Girona for letting me get on board with their paper in the very early days of my PhD.
- the University of York High Performance Computing service, Viking and the Research Computing team for providing and supporting the Viking Cluster, on which I carried out the bulk of my experiments.

The following software has been instrumental in my investigations throughout this thesis and in the communcation of the results in this document.

scipy [4] for statistical tests and the drawing of dendrograms

Savile Row [5] for implementation of SAT encodings, and for the base on which to implement feature extraction and encoding selection

scikit-learn [6] for the implementation of machine learning algorithms

seaborn [7] for quick visualisations to help analyse results

matplotlib [8] for infinitely customiseable plots

List of Figures

List of Tables

List of Algorithms

List of Code Listings

Glossary

arc consistency (AC) Given a constraint C in a constraint satisfaction problem (CSP) and a decision variable $X \in scope(C)$, the variable X is arc consistent if every value in its domain has at least one supporting value in the domain of every other variable in $scope(C)$. xvi, 15

at-most-one (AMO) A constraint across a number of Boolean variables stipulating that no more than one of them should be set to *True*. 13, 133

Boolean satisfiability problem (SAT) A very basic CSP where all variables are Boolean (i.e. can be assigned either a value of *True* or *False*). The single constraint is expressed in propositional logic, connecting the variables with the logical operations NOT, AND and OR. 5–7, 11

constrained optimisation (CO) An umbrella term for various approaches to solving problems with constraints, usually trying to minimise or maximise the result of an objective function. 1, 3, 8, 133

constraint optimisation problem (COP) A CSP which additionally has an objective function which is to be either minimised or maximised. 12

constraint programming (CP) The practice of solving CSPs, which typically involves modelling a problem using a constraint modelling language, and then searching for solutions while using deduction and reformulation along the way to reduce the search space. 1–3, 5, 6, 8, 132

constraint satisfaction problem (CSP) A problem which consists of a set of variables X, their respective domains D and a set of constraints C. Each constraint imposes restrictions on what values individual variables can take (unary constraints) or what combinations of values can be taken by some subset of X. The subset of X which

is constrained by a particular constraint C_i is called the *scope* of C_i. xv, xvii, 3–7, 9, 11, 12, 14, 15, 133

Essence Prime The constraint modelling language used by SAVILE ROW. Essence Prime can be written by human modellers, but it can also be produced by the Conjure tool [9] from problem specifications written in the Essence language. The Essence Prime language is described within the SAVILE ROW manual [10]. xvii, 16, 29, 94–97, 102, 103, 105, 128, 132

generalised arc consistency (GAC) A constraint C has generalised arc consistency if each of the variables in its scope has arc consistency (AC). In other words, for any partial assignment to a variable X in C, there exists at least one supporting value in each other variable in $scope(C)$ for every value in the domain of X. See also AC. 15, 24

graph neural network (GNN) A neural network which is specifically designed to learn from data which can be modelled as a graph structure, such as social networks or chemical compounds. 133

linear integer (LI) A constraint of the form $(\sum q_i x_i) \diamond k$ over the set of integer variables $\{x_1 \dots x_n\}$ with associated integer coefficients (a.k.a. weights) $\{q_1 \dots q_n\}$ and a comparison operator $\diamond$, usually $\leq$ (less than or equal to) 8, 9, 13, 98–100, 103, 104, 106, 107, 111, 112, 115–117, 120–123, 126–130, 132

machine learning (ML) The processing of data in order to build a representation from which useful predictions can be made 2, 6, 8–10, 15, 54, 129–132

multi-mode resource-constrained project scheduling problem A scheduling problem in which tasks can be performed in different modes which affect the amount of resources consumed; the tasks may have precedence constraints and the resources may be either renewable or non-renewable. See [11] 10, 94, 95

PAR10 Penalised average runtime with a tenfold penalty for runs which time out. 49, 62

penalised average runtime (PAR) A metric for the running time of a program. Each run which exceeds a given timeout is assigned a time equal to the timeout multiplied by a predefined multiplier. Several runs are usually executed and their average time reported. 49

pseudo-Boolean (PB) A constraint of the form $(\sum q_i x_i) \diamond k$ over the set of boolean variables $\{x_1 \dots x_n\}$ with associated integer coefficients (a.k.a. weights) $\{q_1 \dots q_n\}$ and a

comparison operator $\diamond$, usually $\leq$ (less than or equal to) 8, 9, 13, 94, 97–100, 102–107, 110–112, 114–117, 120–123, 125–130, 132

SAT encoding A scheme for representing a variable, constraint, or entire CSP as a Boolean SAT formula. The term can refer to the resulting SAT formula, or to the algorithm which produces it. 5, 6, 9, 93, 96, 97, 100, 103, 106, 127–129, 131–133

Savile Row A model reformulation tool, or "modelling assistant" which takes constraint models written in Essence Prime and re-formulates them with a given target solver in mind, applying simplifications and optimisations along the way. SAVILE ROW is described in more detail in [5, 10]. xvi, 8, 11, 29, 96–100, 102–106, 116–119, 123, 125, 126, 129–134

single best The single best algorithm is the algorithm which solves the instances in the training set in the quickest overall time. In this work we are usually referring to the encoding or combination of encodings – this is because SAT encodings are essentially algorithms. 130

Chapter 1

Introduction: Solving Problems by Searching for Truth

♫ Our problems are all solvable ♫

Olivia Rodrigo

Constraints abound in our world. At the time of writing, societies are grappling with the realities of exhaustible natural resources, a diminishing time frame within which to address existential crises and ever more demand on fixed budgets. Increasingly technology is being looked towards to play a major role in addressing some of these universal challenges.

The broad field of constrained optimisation (CO) develops algorithms and approaches which seek to find the best solutions to problems where potentially competing restrictions have to be satisfied by making optimal or at least appropriate decisions with the "levers" that are available. While researchers in this field would never claim to have the answers to all the world's problems, there are many real-life contexts where this research makes a positive contribution to solving difficult problems. Examples of applications from a recent conference in this field[1] include: allocating refugee relief funds for the UNHCR [12], short-term mine planning [13], and strategies for water flow control [14].

Artificial intelligence has never had a higher public profile, but with its ubiquity in modern life come many questions about trustworthiness, fairness and explainability. One significant advantage of CO in general and constraint programming (CP) in particular is that

[1] CP2023 was the most recent constraint programming (CP) conference at the time of writing. Proceedings are available at https://drops.dagstuhl.de/opus/portals/lipics/index.php?semnr=16301. Constraint programming is one specific approach to constrained optimisation.

answers to difficult questions are arrived at by using exact deduction rather than good guesses, and can therefore be explained and justified more easily than approximation-based machine learning (ML) systems such as neural networks.

At the same time, ML can play a vital role in CP to aid decision making during the exact solving process. This thesis considers one particular approach to problem-solving: expressing problems using a constraint modelling language, translating the resulting model into a basic logic formula, and finally solving it using a highly efficient solver. Our focus is primarily on how we can choose translations from constraint models to logic formulae in a way that leads to the problem ultimately being solved faster.

In this introductory chapter I present:

- a high-level overview of the context for my work, namely solving problems using the constraint programming approach,
- the motivation for this work, briefly suggesting opportunities for improving the state of the art in constraint solving,
- the aims of this work stated as research questions,
- the structure for the remainder of this dissertation,
- a preview of the contributions made to the research field by this thesis, and
- some notes regarding what to expect in terms of style and conventions.

1.1 Constraint Programming

When computers are used to solve a problem, there are at least two overarching approaches that can be adopted: imperative and declarative. In the first case, the programmer or software engineer devises a set of exact steps to be taken in order to solve the problem or achieve the required behaviour. The latter method is to capture and communicate clearly the desired features of a solution as well as the known facts and to let the computer come up with an answer. Both ways of tackling complex problems have their potential pitfalls, especially in capturing accurately the requirements or the exact nature and features of the problem at hand. However, by adopting a declarative approach, we hope to benefit from:

- a clear statement of the desired solution, unobscured by “what to do”, and therefore a higher likelihood of actually answering the real problem;

- mathematically sound deduction steps which will give us a reliable solution, reducing the risk of making a mistake in coming up with an algorithm;
- not "re-inventing the wheel", i.e. using existing robust solving algorithms which are applied to our definition of a problem;
- portability of our model not only to different hardware, but more excitingly to different categories of solving technologies.

In this thesis I specifically concentrate on the declarative approach called constraint programming (CP), which is a subdiscipline of constrained optimisation (CO). Constraint programming is concerned with solving constraint satisfaction problems (CSPs). CSPs can model many interesting, important and complex real-life problems ranging from formal verification of hardware systems to allocation of resources, multi-criteria scheduling and layout optimisation. Being a declarative approach, the promise of CP is that a user describes a problem precisely and the computer solves it or declares that no solution is possible. This ambition has affectionately been named the Holy Grail of Computer Programming and progress towards the ultimate goal has been discussed and documented regularly [15].

Let us introduce and illustrate a simple CSP. Imagine a farmer who needs to plant 3 types of crop in their fields in such a way that adjacent fields do not have the same crop. Figure 1.1 represents the layout of the fields (labelled A-G) using edges to show which fields are adjacent on the farm. In this CSP, we model the fields as decision variables A to G, each of which have a domain of $\{1, 2, 3\}$, to stand for the available crops. A possible solution to the problem is the following assignment, which is shown on the right of Figure 1.1:

$$\{A \leftarrow 1, B \leftarrow 2, C \leftarrow 1, D \leftarrow 3, E \leftarrow 2, F \leftarrow 1, G \leftarrow 3\}$$

More specific definitions of CSPs and details of CP will be given later in the thesis. For now it might be interesting to visualise this approach to programming and to compare it with the imperative approach. In Figure 1.2 we see a very high-level view of the CP "pipeline" presented as analogous to the steps taken in imperative programming.

Of course this is a very simplified summary, but it helps us to think about the steps involved and to make some observations of what is important at each stage of the process. In both cases it is vital that the real-world situation is captured accurately in the first instance to ensure that the program actually addresses the real problem. For both disciplines there is potential for creativity in how the programmer uses abstraction to

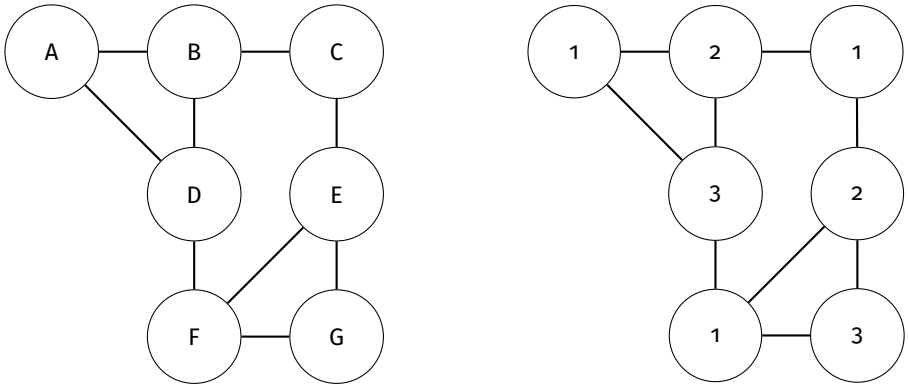


Figure 1.1: Planting crops: an illustration of a simple CSP. Left: a farmer's fields represented by nodes labelled A-G, with edges showing which fields are neighbouring. Right: a possible assignment of crops to fields to avoid using the same type of crop in neighbouring fields.

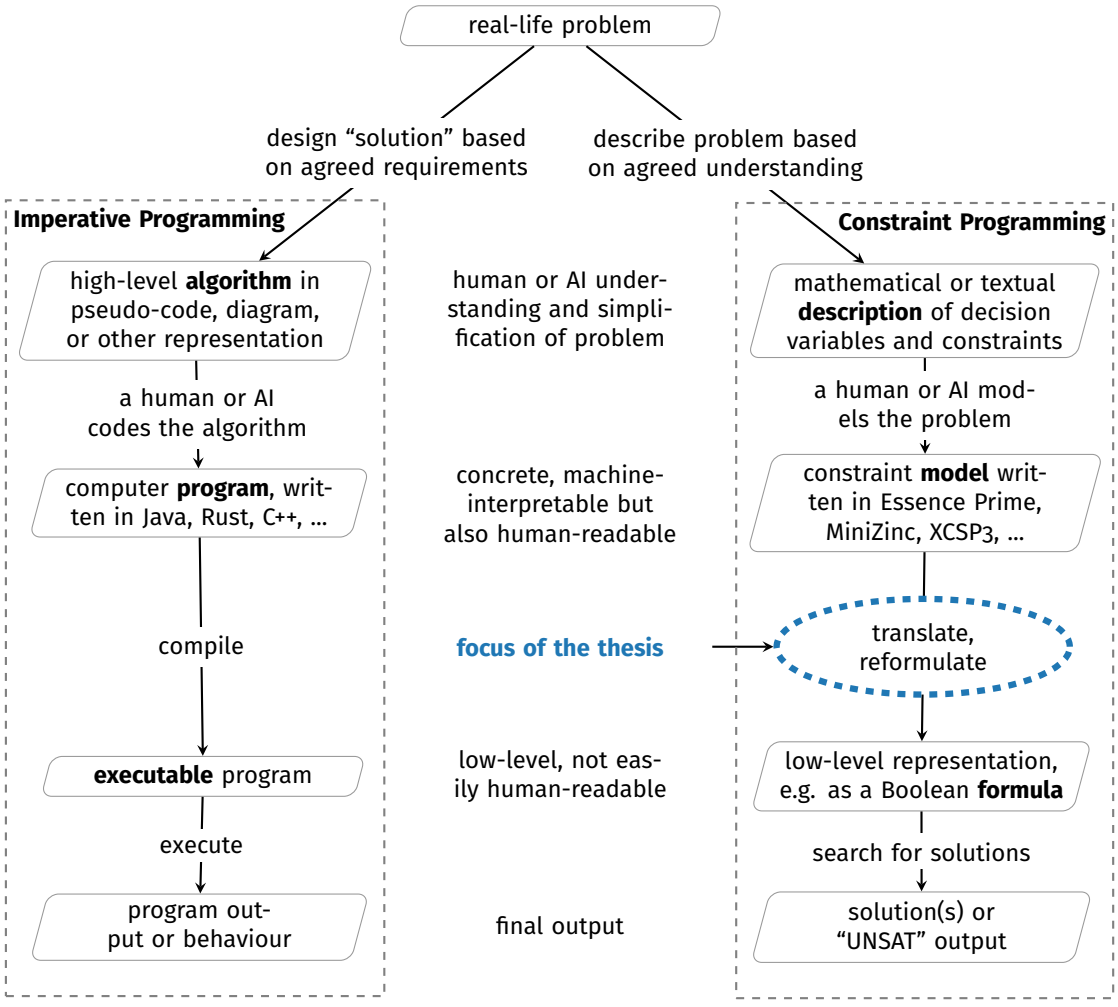


Figure 1.2: An analogy between the imperative programming process and the constraint programming "pipeline", showing a high-level view of how a problem might be tackled using the two approaches. The slanted boxes represent information at various stages. The earlier stages are traditionally carried out by humans, although increasingly AI agents are being developed which can carry out the first two steps with varying degrees of success. The blue ellipse and label highlight where the thesis fits into the process.

represent the situation using their language of choice. The subtle difference is that in CP, the programmer's chief concern is to ensure that the model they produce gives an accurate representation of the features of the problem, whereas in traditional (imperative) programming, the programmer has to engineer the way to achieve the goal too.

We now come to the part of the pipeline which is explored in this thesis. A compiler takes a program written in a high-level language and transforms it into low-level machine code, effectively re-formulating the algorithm in terms of simpler (but less human readable) steps. In CP, the human-readable and expressive model can also be transformed into a simpler, lower-level definition of the problem. This is sometimes called model reformulation, and, just as the details of a compiler can make a huge difference in the efficiency of the eventual executable, reformulation of constraint models can lead to huge improvements in the ultimate solving performance. One reformulation which has proven very effective and popular is to express the constraint model as an instance of the Boolean satisfiability problem (SAT). There are however many schemes which can translate a CSP into a SAT instance. The work in this thesis is mainly concerned with choosing the best scheme for this step of the CP pipeline.

1.2 Motivation

As mentioned above, one popular and increasingly powerful way of solving a CSP is by translating it into SAT. SAT solving technology has advanced apace in recent years and it is a key part of many competition-winning CSP solvers.

Many ways to encode a CSP into SAT have been developed. A SAT encoding for a constraint type refers to scheme which represents the constraint as a Boolean formula. In this context the term "encoding" can refer to the resulting formula (the SAT formula is an *encoding* of the CSP constraint), or more generally to the process by which the translation is achieved, i.e. an algorithm or a mathematical definition. For many types of constraint there exist several alternative encodings. When encodings are described in literature, accompanying experiments often show that, depending on the problem at hand, the choice of encoding can have a significant effect on the time in which the problem is solved, or the memory space requried to reach a solution.

In many areas of computer science complementary algorithms exist for solving a given problem, that is to say the algorithms can outperform each other on different inputs. The idea of using a porfolio of algorithms is well studied and has indeed been applied

with some success to constraint programming but not, to my knowledge, to the issue of choosing a suitable SAT encoding at the constraint level. Users of some constraint solving tools can make the choice of SAT encoding per constraint type explicitly, but no proposed framework or tool exists for making the choice dynamically based on the specifics of the problem being solved.

An experienced practitioner of CP or SAT solving could be expected to consider aspects of a problem and, drawing on experience, make a prediction for which encoding might be the best choice. In the pursuit of the ideal "tell the computer the problem and the computer will solve it", there is a gap in the CP user's toolkit for this expert recommendation to be made automatically. One ubiquitous application of ML is precisely this role of standing in for a human expert to make recommendations. The work for this thesis is motivated by the hope that ML techniques can be employed to speed up the solving of CSPs by making good choices when it comes to which SAT encoding to use.

A further desired contribution is that a thorough investigation of selecting SAT encodings for constraints will lead to new insights and of course new data about the relationship between the nature of the problem being solved and the kind of encoding that works well.

1.3 Thesis and Research Questions

Having identified the research gap and the motivation for this research in the previous section, I can briefly state my overall thesis as follows:

Thesis

The solving of constraint satisfaction problems can be made faster by using machine learning to select SAT encodings for constraints.

The thesis statement above is quite brief for the sake of readability. More specifically:

- The thesis assumes that the problem is to be translated into SAT and solved by a SAT solver.
- The thesis takes constraint satisfaction problems to mean finite integer domain problems, i.e. where the decision variables can take either integer or Boolean values, and each variable has a domain of finite size.
- This thesis restricts itself to studying two types of constraints: pseudo-Boolean and linear integer.

In investigating this thesis, I pose the following related research questions, which are then addressed in the technical chapters.

Research Question 1

To what extent does the choice of SAT encoding for constraints affect problem-solving performance?

Addressed in Chapters 2 and 4

Research Question 2

Can a good SAT encoding choice be made for unseen CSP instances based on generic instance features?

Addressed in Chapter 4

Research Question 3

Can the quality of encoding selection be improved by the use of features which are specific to the relevant constraints?

Addressed in Chapter 5

Research Question 4

Is it practical to learn to set encodings for individual constraints within a problem instance? Does it lead to performance improvements compared to a single encoding choice per constraint type?

Addressed in Chapter 6

1.4 Structure

The rest of this thesis dissertation is structured as follows:

Chapter 2, Background: Constraint problems and Boolean SAT formally defines the types of problems (CSPs) we are dealing with. The constraint solving pipeline is introduced in more detail. Encoding to the SAT is described as one way to solve CSPs. Schemes for encoding specific constraints into SAT are introduced, specifically for pseudo-Boolean and linear integer constraints. We consider literature which shows how the choice of encoding

can affect the solving performance. SAVILE ROW is introduced as a reformulation tool which can support the research carried out for this thesis.

Chapter 3, Background: Machine Learning, Portfolios and Constraints begins with an overview of potentially relevant ML techniques. We go on to explore how ML and CO are being used together to improve solving performance. Finally the idea of portfolios for algorithm selection is discussed, along with existing applications to the CP pipeline.

Chapter 4, Selecting Encodings using Generic Features describes the first attempt to predict good encodings for pseudo-Boolean (PB) and linear integer (LI) constraints using features of the problem instances. The experimental setup is described in detail and the results are presented in a variety of formats to allow analysis. Finally the results are discussed and it is shown that it is indeed possible to predict good encodings on a varied corpus of problems.

Chapter 5, Learning Using Specialised Constraint Features explores whether it is possible to improve on the performance achieved in the previous chapter by devising and using custom problem features which are concerned with the specific constraints whose encodings we are trying to predict. The new features are introduced and the experiments are repeated to evaluate the effect of the new features. Additionally the ML framework used so far is compared to the off-the-shelf algorithm selection tool AUTOFOLIO and it is shown that our custom setup performs better for our task. Finally a measure of feature importance is described and analysed for our featuresets.

Chapter 6, INDICON: Learning to Select Encodings for Individual Constraints advances the research to a more fine-grained decision level, setting a different encoding for each individual constraint (INDICON), rather than a single choice across the instance as before. The chapter begins by setting out some of the challenges that come with trying to learn to set encodings at this level. I describe some methods I designed to gather useful training data. The results of experiments using various setups are presented and evaluated.

Chapter 7, Conclusion evaluates how the findings from each technical chapter relate to the research questions and the overall thesis. Some limitations of this study are explained and avenues for extending or applying this work are suggested.

1.5 Summary of Contributions

The first three chapters introduce the thesis and set out the background to the work. Below is a summary of the contributions made in the three technical chapters.

In Chapter 4, Selecting Encodings using Generic Features:

- the novel design of LeaSE-PI, incorporating ideas from portfolio-building and pairwise predictions
- a detailed empirical investigation and analysis of ML-based predictions of SAT encodings across a varied corpus of CSP instances
- the resulting dataset, published online [2] alongside [3], which includes the problem corpus and all the timing results, as well as the code necessary to reproduce the experiments
- an investigation into the volatility of the "virtual best" encoding due to random elements in SAT solving
- an analysis of the configuration space, i.e. how the available choices of encodings perform across the entire corpus

In Chapter 5, Learning Using Specialised Constraint Features:

- a new set of features measuring aspects of the PB and LI constraints in a problem instance
- a further empirical investigation into the difference between using constraint-specific features and generic features
- an analysis of feature importance
- an empirical comparison of the performance of AutoFolio against the LeaSE-PI specialised setup

In Chapter 6, IndiCon: Learning to Select Encodings for Individual Constraints:

- three methods of obtaining training data for individual constraints
- a novel systematic clustering-based way to extract more fine-grained timing information without trying every single combination of encodings for each constraint

[2] https://github.com/felixvuo/lease-data

- an empirical investigation and analysis of various ML setups for choosing encodings for each individual constraint of a type
- discussion of the opportunities and challenges of this approach, and comparison of the automated approach against the intuitive manual separation of constraints in the multi-mode resource-constrained project scheduling problem (MRCPSP).

1.6 Style and Conventions

Writing Person As stated in the author declaration, I use the pronoun "we" to refer to the author and reader collectively. I use the first-person singular "I" to refer to specific actions or decisions I have made while working on this thesis.

Special Terms I have tried to (re)introduce the full name for each acronym, abbreviation and initialism when first used in a chapter, using the short version thereafter. Special terms such as these are also listed in the Glossary. Some terms used in this dissertation can have different meanings in different contexts. An obvious example is SAT, which can stand for the *Boolean satisfiability problem* in general, or for a given instance of the problem expressed as a logical function of its Boolean variables. I have tried to make the meaning clear from the context.

Epigraphs ♫ These tiny snippets of song lyrics are here purely to make the reader smile, as music has been a faithful companion to me through the often isolated work on this PhD, especially during the pandemic. Please feel free to ignore them.

Chapter 2

Background: Constraint Satisfaction, Combinatorial Optimisation and Boolean Satisfiability

♫ You're yes, then you're no ♫

Katy Perry

The introduction explained why solving constraint problems efficiently is of great benefit in many fields. I now go on to define constraint satisfaction problems (CSPs) more formally and to review established methods of describing and solving CSPs, including by encoding to the Boolean satisfiability problem (SAT). I highlight some of the relevant issues to bear in mind when choosing an encoding by looking at how general constraint solvers and SAT solvers proceed towards a solution.

In this chapter I outline:

- what CSPs are specifically in the context of this thesis
- the structure of a general constraint solver
- a successful method for solving CSPs efficiently, namely by translating to the lower-level Boolean satisfiability problem (SAT)
- how some higher level expressive constraints can be encoded into Boolean logic
- the impact of encoding choice on solving performance
- why Savile Row can help with experiments in this thesis

2.1 Constraint Satisfaction and Optimisation

2.1.1 Formal Definition of CSP

For the purpose of this thesis, I define a CSP as follows.

Definition 1 (CSP) *A constraint satisfaction problem $\mathcal{P}$ is a triple consisting of:*

a set X of n decision variables	$X = \langle x_1, \ldots, x_n \rangle$
a domain $D(x_i)$ for each decision variable, where each domain is either a set of integers, or the set $\{\top, \bot\}$	$D = \langle D(x_1), \ldots, D(x_n) \rangle$
a set of m constraints, each of which imposes conditions on subsets of X	$C = \langle C_1, \ldots, C_m \rangle$

It is often the case that, as well as needing to satisfy a number of constraints, we are interested in maximising some score or minimising some cost. The addition of such an objective creates a constraint optimisation problem (COP), defined more formally below.

Definition 2 (Constraint Optimisation Problem) *A constraint optimisation problem is a Constraint Satisfaction Problem with the addition of:*

- *an objective function mapping the values of a subset of the decision variables to an integer objective value, and*
- *an instruction to either maximise or minimise the result of the objective function*

Definition 3 (Solution) *A solution to a CSP (or COP) is a complete assignment of values to all variables $x_1 \ldots x_n$ such that every variable x_i is assigned a value from its domain $D(x_i)$ and all constraints are satisfied (they evaluate to true). If multiple satisfying assignments exist, then the problem has **multiple solutions**. When no satisfying assignment exists, the problem is said to have **no solution**, or to be unsatisfiable.*

2.1.2 Constraints

Constraints restrict what combinations of values may be assigned to variables. The variables affected by a constraint are called its *scope*.

Practically, constraints can be defined in various ways. Here are a few examples:

- *Unary* constraints have a scope of size 1, applying a restriction on the values that can be assigned to a variable. An example could be that the value must be even.

- *Binary* constraints can be defined for pairs of variables, for example demanding that $x_1 \neq x_2$
- *Extensional* constraints explicitly list all allowable (or conversely disallowed) combinations of values for the variables in their scope – this is sometimes called a *table* constraint
- *Intensional* constraints use a language to describe the restriction, for instance using arithmetic and logical constructs such as $x_1 + x_2 = x_3$
- *Global* constraints are constraints which can be applied to any number of variables. The $\mathrm{AllDifferent}(\{x_a, x_b \ldots\})$ constraint is a global constraint which can be declared on any set of variables to say that their values must be distinct from each other.

In this thesis the focus is on linear integer (LI) and pseudo-Boolean (PB) constraints. We are also interested in at-most-one (AMO) constraints occuring over the variables in a PB constraint. I give definitions below for these three global constraints.

Definition 4 (Linear Integer Constraint) *A linear integer (LI) constraint constrains a set of n integer variables $x_1 \ldots x_n$ so that*

$$\sum_{i=1}^{n} q_i x_i \diamond k$$

where k is an integer constant, $q_1 \ldots q_n$ are integer coefficients (also known as weights) corresponding to $x_1 \ldots x_n$, and $\diamond$ is a comparison operator, such as $=, \neq, \leq, <, \geq, >$.

Definition 5 (Pseudo-Boolean Constraint) *A pseudo-Boolean (PB) constraint is defined over a set of Boolean decision variables $x_1 \ldots x_n$ with associated integer coefficients $q_1 \ldots q_n$ such that*

$$\sum_{i=1}^{n} q_i x_i \diamond k$$

where k is an integer constant and $\diamond$ is a comparison operator.

In effect a PB constraint is identical to a LI constraint whose variables can only take the values 0 or 1.

Definition 6 (At-most-one constraint) *An at-most-one (AMO) constraint is defined over a set of Boolean variables and requires that zero or one of the variables is set to true.*

2.1.3 Solving CSPs

To search for a solution (or all solutions) to a CSP, a solver must assign values to variables from their respective domains and ensure that all constraints are satisfied. In this thesis, we are interested in *sound* and *complete* solvers, by which I mean solvers which guarantee that:

- if no solution is found then no solution exists, and
- if solutions exists, they are all found.

However, achiving the guarantees above can be prohibitively slow for very hard problems, so there are situations where incomplete solvers may still be useful. For instance, it may be sufficient for a majority of constraints to be satisfied. Alternatively, we may be searching for the optimal value in a constraint optimisation problem, but be happy to settle for a result which is "good enough" according to some measure of quality, rather than demanding the provably optimal answer. Although these approaches are of practical use, we do not consider them in this thesis.

A constraint solver typically implements the following actions:

Extend a partial assignment The solver needs to select a $\langle variable \leftarrow value \rangle$ pair to extend the current assignment. Initially no values are assigned.

Check constraints When an assignment has been made or extended, the solver needs to check that all constraints are satisfied.

Propagate implications of assignments When a partial assignment is made, it may be possible to use the contraints in the problem to make further deductions. For instance if a variable we've just assigned a value to is in the scope of an $\mathrm{AllDifferent}$ constraint, then we can remove that value from the domains of all other variables in the scope.

Backtrack If a dead end has been reached, the last assignment must be undone and another value tried. If there are no more values to try at that level, the solver needs to backtrack further and undo earlier assignments to other variables.

The performance of a solver is improved by giving careful attention to the details of how the actions above are implemented. Many heuristics are available for the first action

of selecting a variable and then a value, known as Variable-ordering Heuristics and Value-ordering Heuristics respectively. For example, in [16], a number of variable orderings are trialled at the start of the solving process to train a machine learning (ML) model; this model then periodically switches heuristics by considering a number of features of the ongoing search.

Constraint solvers employ efficient algorithms to update variable domains in light of the latest assignment and thereby reduce the search space for later assignments. This is called *lookahead* and can take various forms. *Custom propagators* are typically implemented for certain global constraints – these propagators use specialised data structures and algorithms to remove values from other variable domains which are incompatible with that constraint. For example, the well-regarded constraint solver Chuffed [17] implements 10 global propagators including *alldiff*, *circuit* and *table*.

Another type of lookahead which has been much researched is the notion of arc consistency (AC). An individual variable $X_i \in scope(C_i)$ is arc consistent with respect to C_i if and only if, when a partial assignment has been made, every value in its domain has at least one *supporting* value in the domain of each other variable in $scope(C_i)$. If every variable in a constraint has AC, then the constraint is said to have generalised arc consistency (GAC). An entire CSP is deemed to have GAC if and only if every constraint in it has GAC. Maintaining AC can be computationally expensive and is not always the most efficient way to search for a solution. As far back as 1994 a sixth variation of an algorithm to implement AC was being proposed and debated in [18]. As we will see soon, arc consistency is still an interesting property when it comes to SAT encodings.

Finally, some constraint solvers can perform the backtracking step intelligently by learning new facts during the search. Again, Chuffed demonstrates this by learning *nogoods* – assignments which cannot take part in a solution. These nogoods can inform non-chronological backtracking (sometimes called backjumping): in situations where a nogood is discovered which does not include the most recently assigned variable, the solver can "jump" over that last decision.

2.1.4 Describing CSPs: Constraint Modelling

We have seen a formal definition of CSPs and had an overview of how they can be solved. However, for CSPs to be practically useful, humans need to be able to express problems easily but precisely. *Constraint modelling* is the process of describing a "real-life" problem (of course this could be a puzzle or a toy problem) in terms of decision variables,

Listing 2.1: A constraint model written in Essence Prime describing the class of Sudoku puzzles. This model is taken from the Essence Prime language description in the SAVILE ROW manual [10].

```
1  language ESSENCE' 1.0
2  letting range be domain int(1..9)
3  given clues : matrix indexed by [range, range] of int(0..9)
4  find M : matrix indexed by [range, range] of range
5  such that
6  forAll row : range .
7    forAll col : range .
8      (clues[row, col]!=0) -> (M[row, col]=clues[row, col]),
9  forAll row : range .
10   allDiff(M[row,..]),
11 forAll col : range .
12   allDiff(M[..,col]),
13 forAll i,j : int(1,4,7) .
14   allDiff([ M[k,l]  | k : int(i..i+2), l : int(j..j+2)])
```

constraints, and optionally an objective function.

A large and growing collection of constraint solving systems exist. Each one tends to require its own modelling interface. This is usually either a *constraint modelling language* or an *API* (application programming interface). In the former case a model is written by the human modeller in the given language and then passed to the solver as input; in the latter case a library is imported and the constraint model is build programatically using a general-purpose programming language.

Popular constraint modelling languages like MiniZinc [19] and Essence [20] allow the user to define a model which describes a whole *class* of problems with parameters for which values can be supplied to create a particular problem *instance*. A basic example would be a Sudoku puzzle – a general model setting out the variables and constraints can be written, but for a particular instance the user needs to supply the pre-filled values.

For this thesis I use the SAVILE ROW modelling assistant [5] which takes models written in the Essence Prime language (previously known as ESSENCE'). An example constraint model written in Essence Prime is shown in Listing 2.1 and describes a standard 9 by 9 Sudoku puzzle. Even in this small example we see some of the expressive power that Essence Prime allows:

- The model describes the whole class of 9 by 9 Sudoku puzzles – the `given` directive (line 3) is used to define parameters of any particular instance.

- Decision variables are defined using the `find` key word (line 4).
- Parameters and decision variables can be expressed as matrices (lines 3 and 4).
- Constraints can be defined using quantifiers variables. For example in lines 6-8 the model ensures that every pre-filled number "clue" is maintained in the result matrix.
- Global constraints such as `allDiff` in lines 10,12,14 allow for more succinct human-readable models.
- Matrix comprehension provide a flexible and powerful way to express constraints within data structures. Our example model uses this facility in lines 13-14 to enfore the rule that each 3 by 3 "box" in our Sudoku puzzle must contain different values.

Savile Row applies various simplifications and reformulations to produce lower-level models specifically optimised for various target solvers. For example it can produce FlatZinc, the same output format as the MiniZinc complier, or target other CP solvers like Minion [21], Chuffed [22, 23] and Gecode [24]. It can also encode CSPs into Boolean formulae for SAT solvers, which makes it particularly useful for my research, as we will see later in this chapter.

2.1.5 Competitions

Competitions are held yearly to promote the development of constraint solving algorithms. These competitions give interesting insights into the state of the art. The competitions described below encourage the research community to submit interesting and challenging problems. These problems may arise from real-life applications or may be theoretically interesting and hard to solve. Teams of researchers enter their solvers (multiple submissions are often allowed) into different *tracks* which place restrictions on either the type of problem or on the solver technology. I report on two such competitions which are well-regarded in the community – recent results from these reveal that a highly competitive approach to solving CSPs is to encode them fully or partially to Boolean formulae which can then be solved by SAT solvers.

MiniZinc Challenge

The MiniZinc challenge [25] has been held anually since 2008 as part of the International Conference on Principles and Practice of Constraint Programming. The exact tracks (categories) have varied over the years but for the last decade the competition has featured

Table 2.1: Constraint solving competition results from https://www.minizinc.org/challenge.html and https://www.xcsp.org/competitions/

	MiniZinc Challenge 2022			XCSP Competition 2022	
Rank	Fixed	Free	Parallel	Main CSP	Main COP
1	OR-tools	OR-tools	OR-tools	PicatSAT	PicatSAT
2	SICStus Prolog	PicatSAT	PicatSAT	Fun-sCOP (cadical)	CoSoCo
3	JacOP	Choco 4	Geas	Choco	Mistral

the following three: the *fixed* track demands that solvers use the search order annotation given in the problem specification (for variable and value ordering), the *free* track allows solvers to use any search ordering, and the *parallel* track allows any search order as well as the use of several processing cores. In all cases the problems are fixed domain CSPs, with all variable domains being Boolean, integer, enumerated, or matrices of these types.

Table 2.1 shows the results from the latest competition at the time of writing. Note that the MiniZinc Challenge winner in each of the three tracks mentioned above is OR-tools [26], a CSP-solving solution from Google. OR-Tools uses a "clause-learning SAT solver as its base layer" according to the description submitted to the MiniZinc 2022 challenge [27]. PicatSAT [28] also employs a SAT solver (Kissat [29]) once it has encoded problems into a Boolean formula.

XCSP3

The XCSP3 [30] language was launched with the intention of facilitating the comparison of algorithms for solving constrained combinatorial problems (CSPs and COPs). Its authors describe it as an intermediate format occupying a space between expressive modelling languages such as MiniZinc or Essence and low-level languages like FlatZinc or SAT. XCSP3 retains some structure of models that is lost when using a flat low-level format.

The authors also organise a yearly constraint solver competition for solvers which are able to take XCSP3 as their input. The results of the competition in 2022 [31] are also shown in Table 2.1. Once again we see that the top places go to solvers which rely on SAT solving technology – in this case PicatSAT and Fun-sCOP (using the CaDiCaL SAT solver).

SAT Competition

The results discussed above indicate that translating a CSP to SAT and handing it over to a SAT solver is a method which is effective for a wide range of problems, given the vari-

ety of benchmark problems submitted to the constraint solving competitions. One final argument for pursuing this route to solving CSPs is that SAT solving technology is itself showing impressive progress. In effect, once we've encoded our CSP as a SAT problem, we continue to benefit from improvements in SAT solving algorithms. Having won the 2020 edition of the SAT Competition [32], the authors of Kissat [29] carried out an interesting investigation[1]. They ran the last 19 competition winners (2002-2020) on the full set of benchmarks from some of the competitions using modern hardware and operating systems. The results showed that the winning solver in 2020 was able to solve between 2 and 5 times more instances than the winner from 2002, depending on which year's benchmark problems were used.

2.2 The Boolean Satisfiability Problem

In order to benefit from the power of SAT solvers, a CSP needs to be translated into a Boolean formula. This process is called *encoding* and any particular schema for carrying out the translation is termed a *SAT encoding*. In this section I define the Boolean Satisfiability Problem (SAT for short), I give a broad description of how SAT solving works, and illustrate some basics of the SAT encoding process.

2.2.1 Definitions

The Boolean Satisfiability Problem (sometimes referred to simply as SAT) is a special case of the Constraint Satisfaction Problem where all the variables are Boolean, i.e. take the value *True* or *False*.

Definition 7 (Boolean Satisfiability Problem) *A Boolean Satisfiability Problem $\mathcal{P}$ consists of a pair $\langle X, F \rangle$ where X is a set of decision variables $x_1 \dots x_n$ each with domain $D(x_i) = \{True, False\}$ and F is a formula in propositional logic made up of variables from X connected with the logical operators AND, OR and NOT. A solution to such a problem is an assignment of* True *or* False *to each variable in X such that F evaluates to* True.

Although a SAT problem can be defined using any formula in propositional logic, it is most common to use Conjunctive Normal Form. This is the input format expected by most SAT solvers, and it supports many of the techniques employed by SAT solving algorithms,

[1] More details and plots at http://fmv.jku.at/kissat/, last accessed 14th March 2023

such as *unit propagation*. The CNF format means that for a solution to a SAT problem, an assignment needs to make at least one *literal* true in every single clause in the formula.

Definition 8 (Literal) *A literal is a Boolean variable which may be negated.*

Definition 9 (Conjunctive Normal Form) *A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses, where each clause is a disjunction of literals. An example formula in CNF is* $(a \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee c)$*; in this formula* a*,* b *and* c *are the variables;* a *and* $\neg b$ *are literals; the expressions in parantheses are the clauses.*

A solution to a SAT problem is either:

- the output "UNSAT", i.e. unsatisfiable, or
- the aoutput "SAT" and an assignment of values to its variables which satisfies the formula

SAT solvers can normally provide proof of unsatisfiability. They are also usually able to output all satisfying assignments for a satifiable problem.

2.2.2 SAT Solvers

Most modern solvers are loosely based on the DPLL algorithm [33] summarised in Algorithm 2.1 – this summary is based on a master class given at the CPAIOR conference in 2020 [34, 35] by Armin Biere, the chief author of a string of recent competition-winning SAT solvers. DPLL searches the assignment space in a depth-first order and backtracks when the current assignment makes the formula unsatisfiable. Each branch of the search tree is expanded as the next unassigned variable is assigned a value. At this point, the implications of this choice are propagated to all affected clauses based on various *boolean constraint propagation* rules, but principally the idea of *unit propagation*.

Unit Propagation

Unit propagation is illustrated in the sequence of steps below.

starting formula	$(a \vee \neg b \vee c) \wedge (b \vee \neg c) \wedge (\neg a \vee b \vee \neg d \vee e) \wedge (c \vee d)$
choose $b \leftarrow F$	$(T) \wedge (\neg c) \wedge (\neg a \vee \neg d \vee e) \wedge (c \vee d)$
$c \leftarrow F$ by UP	$(T) \wedge (T) \wedge (\neg a \vee \neg d \vee e) \wedge (d)$
$d \leftarrow T$ by UP	$(T) \wedge (T) \wedge (\neg a \vee e) \wedge (T)$

In this example, the first line shows the starting formula (the SAT problem) in CNF. In the second line we have chosen to assign a value of *False* to variable b. This means that the first clause is now satisfied and can be simply replaced with *True*[2]. In the second and third clauses b appears as a positive literal; our assignment means that b cannot satisfy those clauses, so b is removed from them. The second clause is now a *unit clause*, having only one "free" literal remaining; in order to satisfy it, we must make c false. The result of this unit propagation is shown in the third line. Now the second clause is satisfied but we also produce another unit clause – the fourth one. We can apply unit propagation again, assigning *True* to d resulting with the formula shown on the last line.

Learning Clauses from Conflicts

A major advance in solving SAT came with the introduction of *conflict-driven clause learning* (CDCL), first seen in the solver GRASP [36]. The key improvement is that when an assignment fails, it is possible to analyse the conflict to determine which variable assignments contributed to the conflict. This allows two new actions. Firstly, a *conflict clause* can be derived (or "learned") and added to the current formula – this clause ensures that the algorithm avoids searching assignment spaces which can never lead to a solution. Secondly, once a conflict has been analysed, it is possible to "back-jump", i.e. backtrack several decisions at once, rather than undoing each decision chronologically. The CDCL algorithm is still based on DPLL but is implemented as a loop, rather than recursive calls. The state consists of the current formula (or clause database), the current assignments and the decision level, which records how many variables have been assigned by choice rather than unit propagation. A high-level summary of the CDCL approach is given in Algorithm 2.2, inspired by Chapter 4 of the Handbook of Satisfiability [37].

Other SAT-solving Tricks

The ability to learn new clauses and backtrack non-chronologically (Algorithm 2.2 lines 8-10) cuts down a lot of redundant search. In addition to these savings, many other advances have also been made in aspects such as:

- efficient lazy data structures for clauses which minimise processing time during constraint propagation
- variable selection heuristics which use information gathered during conflict analysis

[2] We can drop a satisfied clause from the formula but I keep the clauses here for clarity when referring to the nth clause

Algorithm 2.1: High-level overview of DPLL SAT-solving, adapted from [34]. The algorithm takes an initial formula F_0 and returns a result of either SAT or UNSAT; in the former case, it also returns the satisfying assignment

Data: A formula F_0 in conjunctive normal form
Result: Either SAT or UNSAT, and a truth value assignment for each variable in F_0

Function DPLL(F, A)**:**
  $\langle F', A' \rangle \leftarrow$ BCP(F,A) // boolean constraint propagation
  if $F' = \top$ **then** // every clause is *True*
    return $\langle SAT, A' \rangle$
  if $\bot \in F'$ **then** // an empty (unsatisfied) clause exists
    return *UNSAT*
  $\{x \leftarrow v\} \leftarrow$ Select(F', A') // choose an unassigned variable x and a value v
  $\langle r, A'' \rangle =$ DPLL($F', A' \cup \{x \leftarrow v\}$) // branch on $x = v$
  if $r = SAT$ **then**
    return $\langle SAT, A'' \rangle$
  else // did not succeed with $x = v$, try $x = \neg v$
    return DPLL($F', A' \cup \{x \leftarrow \neg v\}$)

DPLL($F_0, \{\}$) // first call with original formula and no assignments

Algorithm 2.2: High-level overview of solving SAT using conflict-driven clause learning (CDCL), adapted from [37]

Data: A formula in conjunctive normal form
Result: Either SAT or UNSAT, and the satisfying assignment in the former case

if *unit propagation results in conflict* **then**
  return *UNSAT*
while *unassigned variables remain* **do**
  carry out "in-processing"
  choose variable and value and increment decision level
  unit propagate
  if *conflict arises* **then**
    analyse conflict
    add any learned clause
    determine which decision level to backtrack to
    if *backtrack level is less than zero* **then**
      return *UNSAT*
    else
      backtrack
return *SAT and current assignment*

- targeted deletion of learned clauses in order to balance the amount of data held (and consulted) in memory with the potential savings in search the learned clauses afford
- re-starting search using a carefully designed schedule, preserving learned clauses

A more in-depth treatment of these aspects of SAT solving are given in the aforementioned Handbook [37].

2.2.3 SAT Encodings

We now return to the crucial step of encoding a constraint satisfaction problem into a SAT problem. Knowing something about how SAT problems are solved should help us to consider what makes a good encoding.

Two decades ago, Bailleux [38] passionately highlighted the importance of choosing good encodings:

> In the challenges that are organized for SAT, only two categories of submission are welcomed: solvers and benchmarks. The issue of improving the proposed encodings in ignored. As the most interesting benchmarks are the hardest ones, ignoring the encoding may have this consequence: some intrinsically easy problems may be made hard by an inappropriate encoding, and, as they are hard, they may be considered as interesting benchmarks. If the final goal is the practical solving of hard real world problems and not only to make solvers overcome inappropriate encodings by rediscovering in the CNF formulas some deductions that are obvious in the original problem, then a careful encoding is crucial and must be considered as a third type of contribution beside solvers and benchmarks in the challenges organized for SAT.

Since then, many papers have been published proposing new SAT encodings for variables and constraints. These encodings are usually evaluated on problem instances inspired by or taken directly from real-life problems.

Variables and Constraints

As explained at the start of this chapter in Definition 1, this thesis only considers CSPs with integer or Boolean domains. A SAT encoding scheme needs to accurately represent the variables and constraints in the original CSP. Although this thesis in effect treats the

encodings as black boxes (an algorithm in a portfolio), it may be useful to appreciate the broad ideas used in SAT encodings in order to interpret the findings of my experiments later on. I therefore give a brief overview below and refer to other literature for more in-depth descriptions of the encodings used in this thesis.

Any Boolean variable in a CSP can trivially be represented by a variable in a SAT formula. To represent an integer decision variable X from a CSP, an encoding must provide Boolean variables and clauses for which there exists exactly one solution for each value in the domain of X.

Arguably the most basic way to encode X into SAT is to create a SAT variable for every possible value in X's domain – the SAT variable is set to true if the integer variable has been assigned that value. To complete the encoding, clauses must be created which ensure that exactly one of the SAT variables representing the variable's domain is allowed to be on. This is called a *direct encoding*. Many alternative schemes exist, such as the *order encoding*, which is based on inequalities, with each SAT variable indicating whether a certain value in the domain has been reached. There are also *log encodings* based on a bit representation of integers, where each SAT variable is true if that bit is equal to 1 in the binary representation of the value. These encodings are described in more detail in [39] – in the same work the authors also show how it is sometimes beneficial to use more than once encoding and introduce *channeling* clauses to keep the representations consistent with each other.

Often the encoding scheme chosen for variables is determined by the nature of the constraints being encoded. For example the *support encoding* [40] operates in the context of binary (pairwise) constraints. Clauses are used to link any value in variable X with any supporting domain values in variable Y, for every combination of X and Y covered by a constraint in the CSP. When encodings are defined for global constraints, a key component is how the variables are encoded into SAT variables and clauses.

We encountered the concept of generalised arc consistency (GAC) earlier in this chapter. Recall that GAC ensures that once a value has been assigned to a variable X, the domains are pruned for all other variables which co-participate in any constraint with X, so that those domains no longer contain any values which could not be used in at least one extended satisfying assignment. Some SAT encodings are able to guarantee that GAC is maintained by unit-propagation. In other words, when a variable assignment has been made to the integer variable X then, once the Boolean variables are set correctly to reflect this assignment, we can apply unit propagation. After unit propagation has been

completed, the SAT variables now represent the pruned domains of the original CSP variables, as defined by GAC. SAT encodings which have this property are said to *UP-maintain GAC*. Referring to Bailleux once again [41], we read that:

> ... all things being equal, the more a solver propagates, the more efficient it is. On the other hand, encodings which UP-maintain GAC generally produce larger formulae than the other ones because they must encode each potential implication of a literal. Of course, larger formulae slow down unit propagation. It is then not always clear which is the best trade-off between the size of encodings and their ability to enforce propagations.

Some SAT encodings have a weaker property, called *consistency checker*, defined in [42] and [43]. This property ensures that at least one clause is falsified via unit propagation if any partial assignment cannot be extended to a satisfying assignment for the constraint.

2.3 Encoding Linear Integer Constraints to SAT

In the introduction I explained that I am focussing on linear integer (LI) and pseudo-Boolean (PB) constraints for two main reasons: their ubiquity in constraint problems and the wealth of encodings designed to translate them into SAT. These factors are underlined by projects such as PBLib [44] in 2015[3]. This library of PB and related encodings allows users to experiment with 6 PB encodings, 3 cardinality encodings and 7 AMO encodings.

2.3.1 Expressing Linear Integer Constraints as Pseudo-Boolean

Another motivating factor is that PB encodings can form the basis of LI encodings. For example, consider the following LI constraint with the integer variables x_1 and x_2 in its scope:

$$2x_1 + 5x_2 \leq 11, x_1, x_2 \in \{1, 2\}$$

The constraint could also be expressed using the following PB constraint:

$$2y_{(x_1=1)} + 4y_{(x_1=2)} + 5y_{(x_2=1)} + 10y_{(x_2=2)} \leq 11, y_i \in \{0, 1\}$$

where the Boolean variables y_i represent values taken by the integer variables. To complete this direct encoding, we would also need AMO constraints to ensure that for each

[3] At the time of writing the most recently maintained fork of PBLib was hosted at `https://github.com/master-keying/pblib`

integer variable only one value in its domain was selected:

$$\mathrm{AMO}(y_{(x_1=1)}, y_{(x_1=2)}) \wedge \mathrm{AMO}(y_{(x_2=1)}, y_{(x_2=2)})$$

and finally a clause corresponding to each x variable to ensure that a value is assigned:

$$(y_{(x_1=1)} \vee y_{(x_1=2)}) \wedge (y_{(x_2=1)} \vee y_{(x_2=2)})$$

2.3.2 Pseudo-Boolean Constraints with At-Most-One Partitions

We saw above one situation where PB constraints together with AMO constraints can provide a useful model. In fact, there are many examples where PB and AMO constraints naturally go together.

As a toy example, let us imagine starting a new job and being allowed to equip our own office. We are working to a budget and have been given a catalogue of priced items we can choose from. We can only choose one desk, one seat and one hardware bundle from the options available. In this case a PB constraint would represent the fact that the sum of the prices of the chosen items cannot exceed our budget. An AMO constraint would partition the available items into categories, allowing us just one pick from each category.

Bofill et al. present a comprehensive study [43] of several SAT encodings for PB constraints which are then optimised for problems where the variables in the PB are also subject to AMO constraints. The experimental section tests these encodings on several problem classes containing PBs with AMOs, including: combinatorial auctions, nurse scheduling and resource-constraint project scheduling.

The key observation in the work cited above is that when AMO constraints are present over the variables of a PB, it is possible to greatly reduce the size of the encoding and therefore achieve big gains in solving performance. The reduction in size stems from the fact that for any AMO partition the maximum value that can be contributed to the total is no longer the sum of weights in that partition but rather the maximum weight. The authors take six existing SAT encodings for PB and "generalise" them to use the AMO constraints if they exist. In addition, they create two new variants of one encoding (the generalized totalizer). I list each encoding below with a brief summary.

MDD This encoding is based on the Binary Decision Diagram (BDD) encoding [45] – BDDs are directed acyclic graphs which end on either *True* or *False*, with the edges representing variable assignments. In MDDs (which stands for Multi-valued Decision Diagrams and were originally introduced in [46], all the variables contained in an

AMO group are represented by a single node, with the edges once again providing a path through to *True* or *False*, one assignment at a time.

GSWC The Generalized Sequential Weight Counter encoding adapts the SWC [47] logic circuit which uses one counter for each coefficient in the PB to build the sum sequentially. In GSWC one counter is used for each AMO group.

GGTd This encoding (and the two which follow) adapt the Generalized Totalizer [48] SAT encoding for PB, which in turn generalises the Totalizer [38] encoding for cardinality constraints. Totalizers are based on binary trees where each coefficient in a PB is represented by a leaf node. SAT variables in the parent nodes represent possible sums of their descendant nodes until the root node is reached which has SAT variables for all the possible totals. The binary tree can be built in different ways. In GGTd the tree is built to be a balanced tree.

GGT In this encoding the binary tree is built using the *minRatio* heuristic, which is described in detail and evaluated in [43]. As the tree is being built bottom-up, *minRatio* is designed to join nodes in a way which minimises the number of distinct sums represented at the proposed parent node – it does this by calculating the ratio between the number of values that a parent would actually require to represent all possible sums and the product of the number of values each of the two children nodes represent.

RGGT The Reduced Generalized Totalizer uses the observation that sometimes there are several values represented at a node of the tree for which any of them would lead to the same ultimate state for the constraint. SAT variables can therefore be used to represent ranges of values rather than just individual values – this can lead to a more compact representation.

GMTO The n-Level Modulo Totalizer (MTO) [49] is a version of the Generalized Totalizer mentioned previously, but the possible sums are represented in a mixed radix base, in theory allowing this encoding to deal with larger integer values more scalably. As with the GGT encoding above, GMTO uses a leaf node to represent all the values in an AMO group.

GGPW The Global Polynomial Watchdog encoding [41] relies on a "watchdog" formula. This formula breaks down the coefficients into their bit values and outputs *True* if an assignment has violated the constraint. This output is generated via unit propagation

after any assignment is made. As with the other encodings GPW is compressed in light of AMO groups and becomes GGPW.

GLPW Devised by the same authors as GPW, the Local Polynomial Watchdog (LPW) encoding follows a similar design (using bit arithmetic) but is more thorough in following up on assignments; the watchdog formula doesn't just signal inconsistency for the constraint, instead it enforces arc consistency for all variables in the PB, setting to *False* any variable which can no longer participate in an extended assignment.

The propagation properties of the encodings above is detailed in the paper; in summary, they all "UP-maintain GAC" apart from GMTO and GGPW. GGPW does have the *consistency checker* (CC) property.

2.3.3 Performance of SAT Encodings

The extensive work on PB(AMO) constraints explored above includes experiments with several problem classes and using two different SAT solvers to compare performance. The results show clearly that different encodings excel for different problem classes. The authors' chosen performance metrics are the solving time quartiles and the number of timeouts for each of the classes of benchmark problem.

In [50] Abío et al. present several SAT encodings for linear integer constraints including modifications to previously designed encodings, for example BDD-Dec which uses a logarithmic encoding of coeffients before building a BDD. In this work, the authors opt for a more unusual way to gauge performance – the *pseudo-harmonic average distance from the solutions found and the best solutions known*. The argument is that their benchmarks are so difficult that often the optimal solution is not found in the allotted time, in which case it is awarded a distance-from-best of infinity. Their distance metric can deal with infinite distances and is also less sensitive to outliers than something like an arithmetic mean.

Both of the investigations above conclude that different encodings seem to suit different problem characteristics. This suggests that a portfolio of encodings is valuable for any general constraint solving solution which uses SAT as the ultimate solver.

Whereas the authors in both cases above provide analysis and commentary regarding which encodings perform well in particular problems, there is no clear or simple guide to selecting encodings for a new problem class. The PB(AMO) paper [43] suggests that one potentially fruitful avenue for further work is "to automatically select an appropriate

PB(AMO) encoding based on properties of the PB(AMO) constraint, such as the number of variables, the magnitude of the coefficients, and the sizes and number of the cells in the AMO partition. Given that there is no single best encoding, and differences in performance are often substantial, an accurate encoding selection method would be valuable". This is a central motivating factor for the work I present in this thesis.

2.4 Constraint Reformulation and Solving with Savile Row

In this chapter we have seen that many constraint solvers exist, accepting different languages to describe constraint models. Some constraint solvers seek to solve the problem end-to-end, while others re-formulate the model and output a problem definition ready for another "back-end" solver to take over.

As explained earlier in Section 2.1.4 I use Savile Row [5, 10] for the experiments in this thesis. Savile Row is named after the famous London street which houses numerous high-quality tailors – this is because Savile Row "tailors" its model to the chosen target back-end for any given execution, for example trying to make use of any specialised propagators available in the target solver if applicable. Figure 2.1 shows the pipeline within which Savile Row operates. The problem is modelled using the Essence Prime language – particular instances of a problem are defined via parameter files or the command line. There also exists a language called Essence [20] which provides a higher level of abstraction through types such as functions, partitions and sets – model *specifications* written in Essence can be translated into Essence Prime models by the Conjure [9] tool. Savile Row can target a range of solver-specific output formats, ranging from classic constraint solvers like Gecode [24] to learning solvers like Chuffed [23], to SAT and MaxSAT solvers [29], and recently even SMT solvers [51].

It is especially helpful that Savile Row implements the eight PB(AMO) SAT encodings we encountered above in Section 2.3.2, namely MDD, GSWC, GGT, RGGT, GGTd, GMTO, GGPW and GLPW. Savile Row also implements its own "Tree" encoding which shares some design features with the totaliser-based encodings. However, the Tree encoding differs in being able to represent integer variables and negative coefficients natively, whereas the PB(AMO) encodings listed above require the constraint to be transformed to a normal form first (details in [43]). The other difference is that Tree is not AMO-aware, i.e. it does not take any advantage of AMO group partitions over its decision variables. More details on the Tree encoding are given in [3]. The encodings above can be invoked for LI and PB

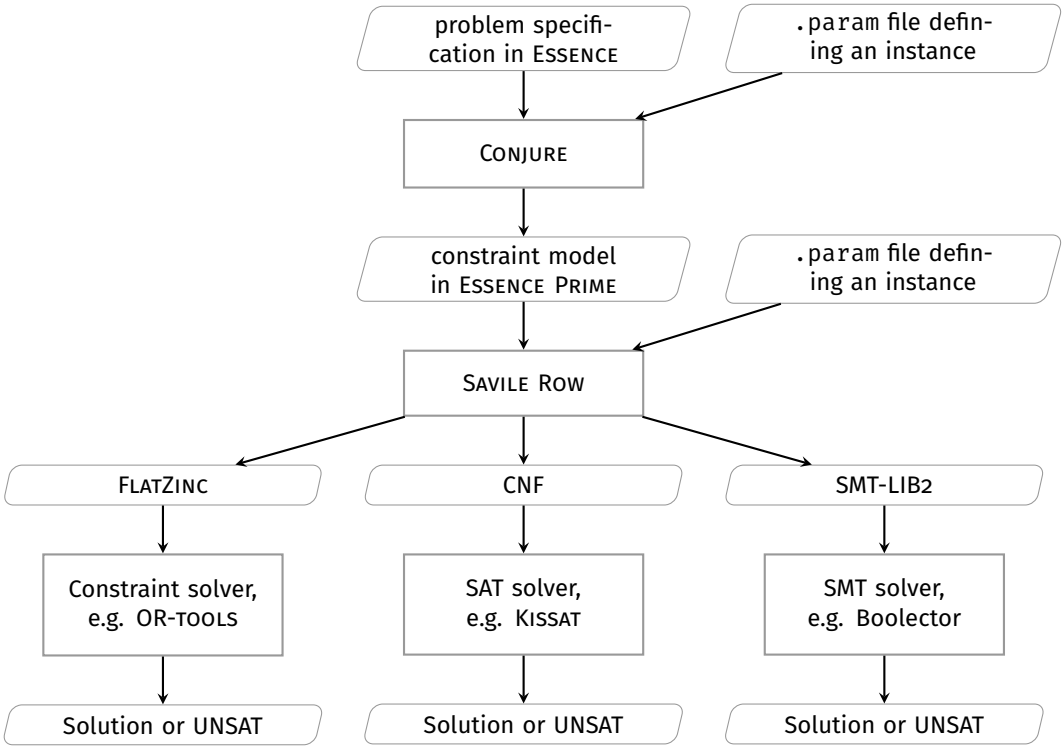


Figure 2.1: An overview of the context within which Savile Row operates. Problems are written in Essence Prime or optionally produced by the Conjure tool. Savile Row produces lower-level models suitable for a range of target solvers.

constraints separately when calling Savile Row with a problem instance, e.g. one can ask for LIs to be encoded using GGPW while using MDD for PBs.

Savile Row offers many optimisation steps. Of particular interest is the ability to auto-detect AMO groups in problem instances [52]. Sometimes AMO constraints are explicitly defined in the constraint models, but it is also sometimes possible to detect such AMO groups implicitly from the whole constraint model. This means that we, the user, can automatically benefit from the more compact PB(AMO) encodings in more instances.

One final pertinent feature of Savile Row is that it can deal with constraint optimisation as well as satisfaction problems. Some target solvers support optimisation directly, so Savile Row can output the relevant objective function. In the case of the SAT back-end, it is not possible to specify optimisation problems, as the only output is a Boolean SAT formula in CNF. Savile Row supports optimisation by iteratively calling the SAT solver with new bounds encoded into the formula, using a bisecting strategy.

2.5 Summary

This chapter begins by explaining more formally some of the key terms and concepts used throughout this dissertation, including definitions of CSPs, COPs, LI and PB constraints. I follow this with a brief overview of constraint programming: how constraint satisfaction problems are expressed using constraint modelling languages and how constraint solvers can search for a solution.

I point out that translation to SAT has been shown to be an effective method to solve CSPs, as evidenced by the types of solvers which have been successful in recent CSP solving competitions. SAT solving as a more specialised type of CSP is explored and some key SAT solving algorithms sketched out. I then briefly introduce how CSP variables and constraints can be encoded into Boolean variables and formulae.

We go on to consider in more detail the specific case of encoding PB constraints into SAT, drawing heavily on the work in Bofill et al.'s paper [43] on encoding PB constraints when AMO groups exist over the PB's decision variables. Here we discover that a wide variety of encoding schemes exist which are complementary in the sense that different encodings perform best for different problem classes.

The chapter concludes with an introduction to the Savile Row the tool which can help us investigate how to make good SAT encoding choices for CSPs.

Having established that many competitive and complementary algorithms exist for encoding LI and PB constraints, the next chapter reviews literature where portfolios of algorithms have been used to make good choices for new problems, especially by the application of machine learning to features of the problem instance being solved.

Chapter 3

Background: Machine Learning, Portfolios and Constraints

♫ I'm hooked into, hooked into machine ♫

Regina Spektor

We have seen that many pipelines exist for solving constraint problems, each with myriad implementation choices. Machine learning (ML) has been employed successfully both in choosing the method used to address a given problem, and to optimise the chosen method. Given the variety of problems which can be expressed with constraint models and the huge effect that choices in the pipeline can have on performance, it is both challenging and potentially rewarding to develop systems which can make good choices reliably.

In this chapter we briefly zoom out to look at how ML and constraint solving can help each other before we focus on efforts which use ML to enhance the performance of constraint solving both in general and in the specific setting of translating CSPs to SAT.

In summary, this chapter covers:

- a very high-level overview of relevant machine learning terms and techniques
- a brief exploration of how ML and constraint solving can be used together
- the use of portfolios for algorithm selection and some examples of systems which use ML to tune the constraint solving process

3.1 Machine Learning

Very broadly speaking, machine learning (ML) refers to the processing of data in order to build a representation from which useful predictions can be made. The data from which the ML system "learns" is called *training data* and the processing of this data is the *training* phase. During this phase, a representation of the data called a *model* is used. The training process can be described as *fitting* a model to the training data. The fitted model can then be used to make predictions about data in the *testing*, *evaluation* or indeed *production* phases. Generally the intention is that the ML model can *generalise*, meaning it can make useful predictions for previously unseen data. This fails if a model is fitted too closely to the training data – this phenomenon is called *overftting*, and many techniques have been developed to avoid it. The training and prediction rely on a set of *features* for each item in the data – this set of features is typically a vector of numerical values but could be any collection of measurements or observations.

3.1.1 Types of Machine Learning

Machine learning can be grouped into three main types and all are potentially relevant to our domain of constraint satisfaction and optimisation.

Supervised Learning uses labelled data to construct its model. The examples in the training data each have a label and the ML algorithm can build its model to predict the correct label for new data points. This label could be categorical, in which case the task is called *classification*, or it could be a value in a *regression* task. An example could be sentiment analysis from a piece of text, i.e. determining whether the text is positive, neutral, comic, threatening, etc. In the training data, each piece of text is labelled with the relevant category. When a new piece of text is presented, the ML model predicts which label is most appropriate. A further application of supervised learning is for *ranking*; in some ways this task is akin to both classification and regression in the sense that we are dealing with distinct items but are looking for a prediction which tells us how those items should be ordered according to some desirable criterion. An application of ML ranking could be in assisting medical triage [53].

Reinforcement Learning employs the idea of rewarding actions. The system takes a sequence of actions and a reward function is used to calculate the score either after

each individual action or at the end. The model which is developed in this approach is called a *policy* and it determines what action should be taken next given the features of the current *state* and what it has learned so far from the rewards it has received. Reinforcement learning could for instance be used in a robot to feed back how well it is doing in attaining a goal, such as navigating a room full of obstacles. The features at each stage could be its sensor data and the reward could be made up of whether it has hit something and how close it is to its destination.

Unsupervised Learning describes the search for patterns or groupings in data without outside intervention or guidance. This approach seeks to find out to what extent the data can be organised into clusters which share similar features in some way. One potential application could be splitting a team of athletes into a number of groups based on their physical and fitness readings in order to design different workouts.

Deep Learning As well as an excellent and concise explanation of the ML terms described above as part of a survey of the applications of ML to combinatorial optimisation, Bengio et al. [54] also dedicate a separate entry to *deep learning* [55], which can be applied to supervised, unsupervised or reinforcement learning. In deep learning, a network of *neurons* (nodes) is organised in layers, with values being output by one layer of neurons and becoming inputs into the next layer. Each neuron processes the inputs subject to its own parameters and outputs a value after applying an *activation* function. Each node's parameters (usually a *weight* for each input, and an additional *bias*) are learned by repeatedly adjusting them in response to a score calculated on the output of the final network layer. This adjustment is typically done using *back-propagation*, an algorithm based on partial differentiation.

In some scenarios it is very difficult or expensive to obtain sufficient labelled data to train an accurate model using supervised learning. This has led to the development of further types of ML, which are a hybrid of supervised and unsupervised learning, such as:

Semi-supervised Learning can be applied when many training samples are available, but only a small proportion of them are labelled. A recent survey [56] of semi-supervised learning techniques provides a taxonomy to organise the many approaches which can fall in this category. The authors' chief distinction is between inductive and transductive methods. In the former case unlabelled data is used in addition to labelled examples in the training of a model, for instance by training a classifier on

labelled data and then using that classifier to predict labels for unlabelled data, producing "pseudo-labelled examples" which are then used to train an eventual classifier. In the latter set-up, unlabelled data are assigned labels according to their proximity to labelled examples and the whole set is then used in training. Of course, many assumptions are made and this type of learning can only be applied effectively under certain conditions.

Self-supervised Learning is a subclass of unsupervised learning where something is learnt from the data without having a task in mind. The model thus trained has picked out potentially relevant features of the data and can then be used to carry out a specific classification or generation task. This approach has been used in image classification through the use of "pretext" tasks [57]. A more general overview is given in [58] where self-supervised techniques are described in additional contexts such as natural language processing and graph learning.

3.1.2 Designing a Machine Learning System

Let us consider some of the elements which make up an ML system – these aspects may need careful consideration when it comes to applying ML to the problem of selecting good encodings for CSPs.

Which ML Agorithm

The most important choice to be made, in a sense, is which ML *algorithm* to use. Or, at least, we can say that it is helpful to make this choice early when designing an ML pipeline because the choice of algorithm could impact other aspects, such as how the features should be pre-processed, or the hardware that might be required to run the algorithm.

The choice of ML algorithm determines the type of *model* used, i.e. the kind of data structure that will represent the data. During training, a *fitted model* is produced which represents what has been learned from the training data. During the rest of this dissertation, I generally use the word model to mean a fitted model. Some ML algorithms do not create any separate model; instead, they use the existing training data to make new predictions.

Here is a brief overview of some popular ML algorithms for **supervised learning**, all of which can be used for classification or regression:

Decision Trees [59] are models which, once constructed, present a path of logical tests from root to leaf which yield the predicted label or value for the features presented. To construct decision trees, a recursive algorithm selects a variable and a value pair (initially for the root) which is most discrimating with respect to the target labels. This evaluation is usually done on the basis of statistical impurity or entropy. Decision trees are easily interpretable and relatively fast to train, but they can suffer from overfitting to the training data, i.e. not generalising well. Some techniques can be used to mitigate overfitting, such as limiting the depth of the tree, or insisting that a leaf node must match a minimum number of items in the training set.

Forests of decision trees seek to address the overfitting tendencies of single trees by training several trees. At prediction time, each tree "votes" for a solution, and these votes are aggregated in some way to produce an overall prediction. The most famous algorithm using such an arrangement is *Random Forests* [60], which employs two random elements: each tree is generated using a bootstrap sample of training examples, and random subsets of features are used when constructing the trees. Another competitive arrangement is a forest of *Extremely Randomised Trees* [61], where the attribute to branch on and the threshold values are chosen at random. Forests of trees retain some advantages of using decision trees: the input data need not be scaled, and the features can contain different data types. Interpretability becomes more difficult as the decision is made from averaging the votes of individual trees.

Gradient Tree Boosting iteratively builds a sequence of trees which are retrained at each stage in a way which focusses attention on previously misclassified entries. This is done by training on "pseudo-residuals" of the training data – these residuals are obtained from the derivative of the loss function. Full details are given by Friedman in [62] along with further enhancements, such as random perturbations at each stage leading to more robust predictions. A very fast implementation of boosted trees is XGBoost [63] which has been popular in the last decade, and often recommended as a good first tool to try on a problem.

K Nearest Neighbours is a technique which relies on the actual training data rather than building a representation. In order to predict the label or value for a new item, this algorithm finds the K nearest examples in the feature space and returns an average of their labels.

Support Vector Machines [64] map the training data into a higher-dimensional space and then calculate a hyperplane which achieves the largest separation between two groups of points. These decision boundaries are stored as a small set of support vectors, which makes it a memory-efficient algorithm. Different kernel functions can be used for the mapping – this versatility can be used to tailor the support vectors to different input distributions.

Multi-layer Perceptrons are the basic implementation of *artificial neural networks* and the basis for the idea of deep learning, which was introduced above. MLPs consist of an input layer, an output layer and at least one *hidden layer*. The number of nodes in the input and output layers tend to be related to the number of features in the input and the number of possible labels in the output respectively. Choosing an effective architecture (i.e. how many hidden layers of what size) for such a network is a complex design problem, and generally neural networks tend to require a large amount of input data to converge on a model. However MLPs have been shown to be effective for many classification and regression tasks and offer immense flexibility.

Ensemble methods combine several separate predictors to arrive at a final output, the idea being that different component estimators may have complementary prediction strengths across the expected distribution of inputs. Random forests are an example of an ensemble; however ensembles can also be made up of different classes of predictors.

The supervised ML algorithms mentioned above are summarised in Table 3.1, which highlights some key advantages and disadvantages of each algorithm.

In addition to algorithms for supervised learning as outlined above, I also use unsupervised learning in this thesis. Recall that in unsupervised learning, there are no labels with the data to indicate what the right value or class is. Instead the user seeks to organise the data in some way depending on the characteristics exhibited by the data. Below I give a brief introduction to some potentially useful **unsupervised learning** algorithms.

Principal Component Analysis is a method for representing data in fewer dimensions while retaining as much information as possible. It operates by finding linear combinations of variables and ranking them by how much they explain the variance of the original data. PCA can be useful for visualising multi-dimensional data or for reduc-

Table 3.1: A summary of selected ML algorithms and some potentially relevant considerations

Model	Advantages	Disadvantages
Decision Trees	explainable, quick to train	prone to overfitting
Random Forests	features don't need scaling	explainability
Gradient boosting	can converge quickly	may overfit for noisy data
K Nearest Neighbours	simple to implement, fast (no training), few hyperparameters	models grow very large
Support Vector Machines	memory efficient	can be slow to train on large datasets, can be difficult to pick good kernel
Multi-layer Perceptron	very flexible, can find complex patterns	many architecture design choices

ing the number of features in order to speed up the operation of another learning algorithm.

K-means Clustering assigns data points to a pre-defined number of clusters K. Initially K *centroids* are randomly generated in the data space. Iteratively, each data point is assigned to its nearest centroid, after which the centroids are updated to the mean location of their assigned members. The process is repeated until the centroids effectively stop moving between iterations.

Agglomerative Clustering begins by treating each datum as a cluster. A neighbourhood radius is repeatedly increased so that clusters join together when they are within a given distance from each other. There are different ways to measure the distance between clusters and indeed different ways to calculate the nominated location of a cluster from the locations of its constituent members in the feature space. With this method, it is possible to either pre-determine the number of clusters required, or to examine the jumps in distance where clusters join in order to decide how many clusters to use.

Autencoders are a class of artificial neural network which can be used to create a lower-dimensional representation, or *embedding* of the training data. At its simplest, the network contains output nodes which match the input nodes exactly, but a smaller internal layer, which represents the embedding of a given input. The network is trained to make the output match the input exactly; the idea is that any structure inherent is the data is learned so that the original input can be recreated from a

compressed representation. This architecture can for instance be used to address noisy data, or to reconstruct an example when there are gaps in the data.

Splitting Data for Training, Tuning and Testing

I made the distinction earlier between the training, evaluation and production phases of an ML project. The data used at each stage is vital:

- During **training**, the data used should be varied enough to allow the ML algorithm to be exposed to as much of the relevant feature space as possible so that it can be well-prepared to return a prediction for new unseen examples.
- During **testing** or **evaluation**, the data should be distinct from the training data so that we can judge whether our system is making good predictions, rather than just returning a previously seen "answer". At the same time we would be expected to test on examples that come from the same underlying distribution as the training data and ultimately the expected live or production data.
- By the time an ML system is in **production**, its designers, operators and users would need to be satisfied that the system has been trained to work well with the type of data it faces in the live setting.

The statements above are open to a lot of interpretation and debate which would need to take place in the context of a specific application. There are many deployment scenarios where failure to properly consider the data used at all stages of design could result in anything from inconvenience to death. The safe and ethical deployment of ML systems is very topical, and a lot of the analysis is focussed on the type of data used to train and test such solutions. An interesting framework for thinking about the assurance of ML systems in more general terms is provided in [65], where the authors argue specifically that data should be managed to "help ensure that ML training and verification datasets are Relevant, Complete, Balanced, and Accurate."

Especially in the context of academic and scientific exploration, it is expected that we judge the effectiveness of a system as objectively as possible. It is therefore necessary to be clear about how training and test examples have been separated during the development of a solution. In our case, both training and test instances are to be drawn from a subset of the general distribution of constraint satisfaction problems. Once such a relevant corpus of problems has been assembled, it is important to ensure that the test

data is not seen at all during the training phase. Should such *data leakage* occur, it would damage the reliability of any conclusions made from the testing phase.

ML-related literature often mentions *validation* data. Although this is sometimes used as just another name for test data, it is in fact usually referring to a split made within the training phase. Most ML algorithms have hyperparameters that can be set; for example how many neighbours to consider in a K-nearest neighbour classifier. It is common to set some of these hyperparameters by automated *tuning* – for this purpose, the training set is itself separated. One popular way to do this is called K*-fold cross-validation*; it splits the training data into chunks, say five chunks A-E. The algorithm would be run with certain hyperparameters to build a model trained on data in A-D, and then used to predict labels for chunk E. This split would be repeated 5 times, with each chunk playing the part of a test set. The scores from each of the 5 iterations, or *folds* would then be averaged to give the overall score for that particular configuration of hyperparameters.

Features

Modern ML systems can operate on a wide variety of data ranging from audio recordings to tabular numerical data to infra-red images to graph structures. Each data point is usually represented as a set of features – these could be the pixels in one image or the sensor readings at one location at one time. Traditionally, and in the case of most of the ML algorithms summarised earlier, features take the form of rational numbers, although some algorithms (such as decision trees) can work well with discrete integers, Boolean values or categorical data. It is often necessary to pre-process data sets in order to make them suitable for a particular algorithm. Some relevant aspects are introduced below.

Cleaning Some ML algorithms cannot handle missing values in any of the expected features, in which case it is important to deal with such data points. One can choose to exclude items with incomplete information, although in some circumstances this would waste a lot of precious other information. More commonly, a value is *imputed* for a missing feature. Imputation calculates a reasonable value by considering the distribution of values in the rest of the data set for that particular feature – this value could be an average, or simply a random sample from the rest of the distribution.

Scaling or Encoding It is sometimes necessary to scale feature values in order to achieve sensible results – for instance if the features “number of children” and “value of house”

were both supplied to an algorithm which represented data points in multi-dimensional space, the influence of the two features would be very unbalanced. Common techniques include linearly scaling all values between 0 and 1, or normalising a feature's distribution to end up with a mean of 0 and standard deviation of 1. Categorical values sometimes need to be encoded as numbers, which can be done by a sparse "1-hot" encoding, where each possible category is represented by a Boolean variable, and a *true* value means the item belongs in that category - this also means multiple categories can be applied to one example.

Selection A valuable pre-processing option is to reduce the number of features presented to the algorithm by considering whether all raw features actually provide useful information. In some cases, it may be that two features are so tightly correlated that removing one of them has no impact on the prediction capability of the trained model. For some algorithms, a smaller number of features means faster training times. Fewer features may also avoid an algorithm learning from features which ultimately are not so relevant. There are many approaches to feature selection, including using correlation metrics to identify features which in effect give duplicated information. Another approach is *dimensionality reduction* where the data points are projected to a lower-dimensional space, taking care to keep data points distinct from each other.

Importance It can be useful to measure or rank the importance of features in terms of their contribution to the quality of prediction. If we can obtain this information, we can use it for instance to do feature selection, focussing only a subset of features with high importance. It may also be useful in explaining how the ML system reaches its conclusions. Finally, an idea of which features are important could feed back to the very start of the lifecycle of a project and place a greater emphasis on obtaining accurate values for those features. Some algorithms can produce feature importance values as part of their normal operation, for example decision trees have to calculate which feature is most discrimating at every node. Other techniques also exist, such as *permutation feature importance* (see the section titled "exporing the random forest mechanism" in [60]) which shuffles the values of one variable at a time, and measures the related degradation in prediction accuracy. A more comprehensive technique called *sequential feature selection* [66] considers subsets of features by iteratively removing one feature at a time from the full feature set or by growing the feature set one at a time. In either orientation, the result is a set of

features beyond which the performance starts degrading or respectively stops improving beyond a pre-set threshold.

Extraction A final consideration is how the features are obtained in the first place. In some cases this is obvious, e.g. once an image exists, its features are the individual pixels. In other situations, useful features may need to be calculated. For example, if the item under consideration was a graph, we may want to extract some information about the graph such as the average node degree. A compromise needs to be struck between how informative the features are and how long it takes to compute them – when dealing with large graphs it can be computationally expensive to extract certain useful features. This balance applies in many settings, not just with graphs.

Hyperparameters

Almost all ML algorithms allow for design choices to be passed in as hyperparameters. Examples include the number of clusters to create using K-means clustering, the maximum tree depth for a random forest, and the configuration of hidden layers in a neural network. These general-purpose ML algorithms can be made more effective in the application scenario by selecting hyperparameters well.

Although some heuristics can be applied from shared good practice in literature, it is common to *tune* the hyperparameters by repeatedly training one type of ML model using with different hyperparameter values. To avoid depleting the full dataset, hyperparameter tuning is often done using a technique called k-fold cross-validation (CV), introduced above in the discussion on splitting data. This technique allows us to test out one configuration of hyperparameters and obtain a performance score. The search for good hyperparameters can thus be automated; the hyperparameter space can be explored in different ways including a basic *grid search*. In this approach, the user provides a finite set of candidate values for each hyperparameter, and k-fold CV is used to score each possible combination systematically. This can obviously be very time-consuming, especially if the underlying ML models take a long time to train. An alternative approach to finding a good hyperparameter assignment is to use randomised grid search, where values are sampled from each candidate set and a fixed number of trials are carried out. Even more sophisticated is the sequential halving approach [67] in which increasing training budgets are allocated as unpromising configurations are discarded.

An important element of hyperparameter tuning is the choice of *scoring function*. Most

ML algorithms have a default metric for the quality of predictions on a test set. For example in the `scikit-learn` library, classification algorithms default to `accuracy_score` which is the fraction of samples correctly classified, and for regression the default is the coefficient of determination or R^2. In some settings, the user may wish to optimise the hyperparameters using a more specific application-inspired score. Perhaps misclassification can be said to have degrees of significance. For example if an ML system predicts paths through a graph it could predict a non-optimal path whose length is in fact quite close to the best one. In this case we might want our scoring function to represent this near-miss with a better score than a path which was much worse.

Hyperparameter tuning can also in effect suffer from overfitting – the tuning process, whether manual or automated, results in a decision of what particular features the ultimate model should have, e.g. how deep the trees should be in a random forest. This decision is made on the basis of the training data available and it's therefore possible that the hyperparameters chosen lead to models which are much better for the training data than the production data they will need to operate on.

Some Pitfalls

Finally in this section, we draw out some of the main dangers to watch out for when creating an ML system.

Overfitting If we train our model too well on the training data, in particular on some of the noise that may be present, we run the risk of creating a model which does not generalise well to unseen examples. Many tactics exist to mitigate this risk, including

- ensuring a diverse training set,
- avoiding overspecialised models, e.g. by using dropout (not fully connecting all nodes) in neural networks, or restricting our forest to using shallower trees,
- using an ensemble of different types of model.

Unbalanced data If in the training phase the distribution of labels is uneven, the model may not learn well. For example, if 99% of training data shows healthy plants, then an ML classifier trying to predict disease in plants will get excellent accuracy scores by simply predicting health each time, but this would be a useless system. Some ML algorithms are

more susceptible to poor predictions in this scenario. It can be addressed in various ways, for instance by

- curating the training data carefully to achieve good balance of target labels,
- over- and under-sampling items with the minority and majority labels from the training data,
- synthesising extra examples for a particular category entirely artificially, or by perturbations to genuine data.

Opaqueness or Lack of Explainability Some ML algorithms are completely white-box in the sense that it is easy to interpret how a trained model produces a prediction, as is the case for a basic linear regression or for a decision tree. In the majority of cases, however, such explanations are impossible to extract completely. There are some techniques which go some way towards understanding how an ML system reached its decision. This differs depending on the details of each algorithm, but for instance it may be possible to trace the contribution of an input value through a neural network to see to what extent it influences the overall result. Another contribution to explainability is to extract feature importance information – the ranking of variables can at least point the user to which parts of the input most affect the overall result. An interesting survey of efforts towards explainable artificial intelligence (XAI) in more general terms can be found in [68].

Following this broad overview of ML systems and the various aspects which need careful consideration when designing an ML solution, we now look at how ML has been applied to the process of solving constraint problems.

3.2 Using Constraint Solving and Machine Learning Together

Figure 3.1 illustrates the different arrangements in which machine learning and constraint solving have been described in the literature we refer to below.

3.2.1 To Augment or Replace?

Kotary et al. [69] describe two main senses in which machine learning and constrained optimisation[1] (CO) can co-operate to advance the performance of automated problem-

[1]Constrained Optimisation is a more general term encompassing constraint programming and other related technologies which seek to find a solution to a problem involving decision variables and constraints imposed on those variables

solving.

In *ML-augmented CO* machine learning techniques are used to make choices as part of an established constraint optimisation algorithm. Any arbitrary choice can be made with reference to an ML model. Some examples are: guiding search by determining the variable and value order, deciding when to re-start search, or choosing which learned constraints to "forget".

On the other hand, in *end-to-end CO learning* the aim is to build ML systems which learn how to solve CSPs/COPs entirely, or at least to find very good approximate solutions. This approach is described by Vesselinova et al. [70] in the domain of graph-based combinatorial optimisation. The authors summarise several works which use either supervised learning or reinforcement learning to predict solutions. In many cases the solution quality approaches or exceeds exact methods.

One additional set-up explored in Kotary et al.'s survey [69] is to employ traditional constraint solving as a layer of a neural network providing "structured logical inference". With the explosion of applications of ML across every imaginable sector of society and industry, it is becoming increasingly important to provide guarantees regarding the output of ML systems. Constraint solving can play a role in ensuring that certain conditions have to be met by the outputs of complex ML systems, especially where a combination of outputs is produced. The permitted output combinations can potentially be modelled with constraints. A fun example is Lafleur et al.'s work[71] which modifies a music generator based on deep reinforcement learning by adding a constraint programming step which ensures that output obeys music theory rules.

3.2.2 Tuning versus Collaborating

On the subject of applying ML to combinatorial optimisation, Bengio et al. also present a survey of techniques [54], similarly categorising the approaches broadly into *end-to-end learning* and *ML-augmented CO*. They subdivide the second into *learning to configure algorithms* and *ML alongside optimisation algorithms*. This latter division makes the distinction between ML being used once outside of the CO algorithm, or being employed during the execution of a CO algorithm, having access to the intermediate state so as to inform next steps in the CO solving process.

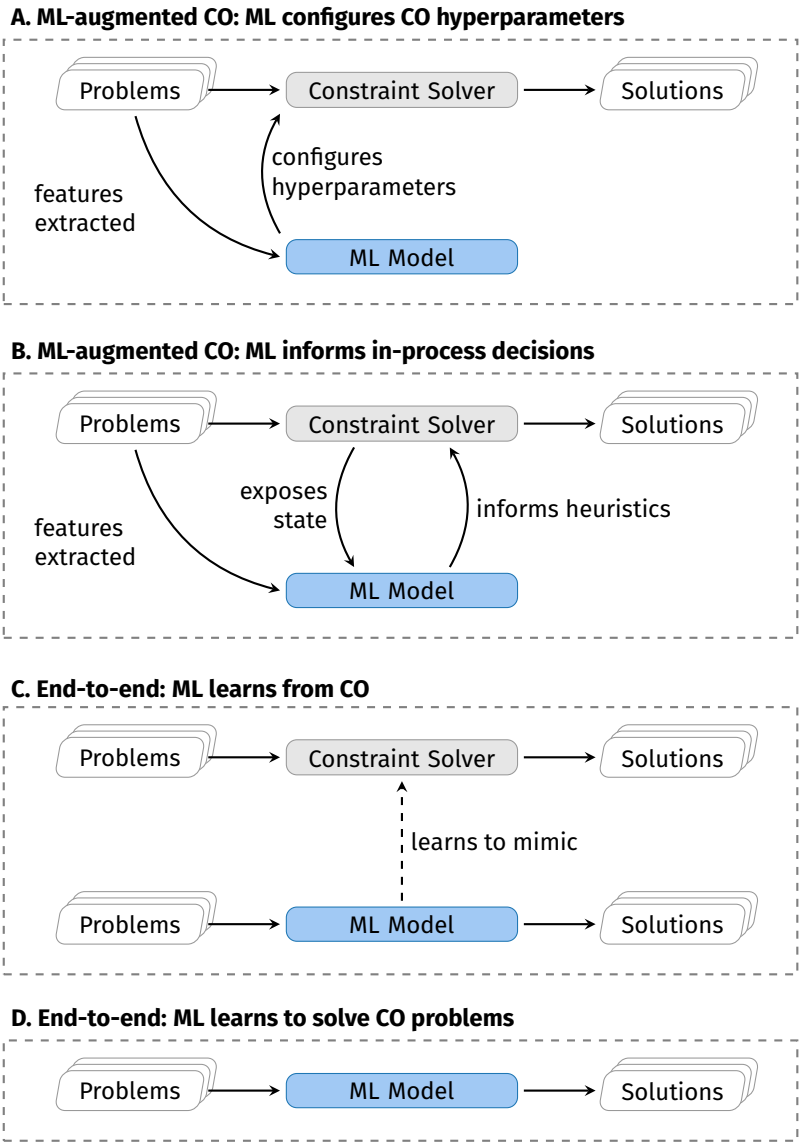


Figure 3.1: Different arrangements for the use of ML in solving CO (constraint) problems, as presented in the cited literature. A: ML configures the hyperparameters based on features of the problem instance at hand. B: ML uses features of the problems and information about the solver's state to inform heuristics as search proceeds. C: An ML model has learned to mimic the solutions of an exact algorithm from past solving runs. D: An ML model has been supervised to learn to solve CO problems directly.

3.2.3 Copy the Expert or Discover a Policy Independently?

The authors cited above [54] make a further interesting disctinction when it comes to the ML itself. The two cases considered are when an ML system tries to mimic the choices of an expert or oracle versus an ML system learning to make decisions based on an eventual reward, essentially using the reinforcement learning (RL) paradigm. The former approach is supervised learning where the ML is trained on choices which are deemed to be optimal or desirable based either on empirical or theoretical expert knowledge. In the latter case, a reward function is constructed which encapsulates the desired characteristic of

the system, for example good quality solutions or a fast solving time.

This thesis is concerned with making good choices when it comes to using SAT encodings of constraints. This would be expected to fit broadly into arrangement A in Figure 3.1, setting configuration options for the constraint solver based on information about the problem instance. There might also be a way for the ML to receive further information from the constraint solver to inform its recommendation of SAT encodings, which would place it closer to the second arrangement, i.e. providing support during the constraint solving process even after the solver has been called.

3.3 Portfolio Approaches, Algorithm Selection and Configuration

In our particular situation, we are essentially looking to choose from a menu of options, rather than the more general idea of hyperparameter tuning, where the decision space could include continuous numeric variables. A lot of success has been reported for systems which "choose the best tool for the job", drawing from a portfolio of available algorithms to attempt to solve a given problem as quickly or well as possible. This process is called *algorithm selection* and is explored and formalised by Rice in [72].

We could also consider our task as *algorithm configuration* [73], where a constraint solving algorithm exposes the choice of SAT encodings as a parameter to be configured or tuned.

The thesis draws from both approaches, considering our setup as an algorithm configuration challenge where the algorithm in question is called solve a CSP/COP. Inside this algorithm we can delegate the task of encoding constraints into SAT to any one of a number of available schemes which are themselves algorithms, thus enacting algorithm selection.

3.3.1 Building a portfolio

For any type of task there may be a vast range of algorithms to choose from, but it may be inpractical or undesirable to employ them all when constructing a solving pipeline for the task. Practically, it may be that effort is required to prepare data in a way that each algorithm can consume, or that some of the algorithms have fairly similar competence profiles, or indeed that some algorithms might dominate others for all the intended tasks, i.e. outperforming them in all conceivable situations, therefore making some algorithms

redundant.

Construction

Muñoz et al. [74] investigate and evaluate 5 algorithm construction methods:

Random K simply selects at random K algorithms from the available pool

Top K chooses the best K all-round performers, i.e. the algorithms which solve the most problems in the training set within a given time limit

Sequential Forward greedily adds one algorithm at a time such that each addition yields the largest reduction in runtime; this technique can be run until a fixed portfolio size is achieved or until the potential runtime is within a given threshold of the absolute best possible runtime from the entire available pool

Sequential Backward is similar to the previous approach, but eliminates one algorithm at a time from the portfolio until a target size is reached, or the performance degrades too much

Mixed Integer Programming Here the construction problem is framed as a MIP problem, which seeks to minimise the average maximum regret of chosen items.

The authors try out these techniques on problem instances from the ASLIB library [75], which contains a wide variety of algorithm selection scenarios. They conclude that the Sequential Forward technique appears to be the best for balancing both overall performance in terms of low regret and best robustness.

Complementarity

It is clearly desirable to build a portfolio which is robust, i.e. solves most of the problems likely to be encountered in a reasonable time. However, when it comes to designing the candidate algorithms, this has not necessarily been foremost in the minds of the designers. Solver capability is often tested and recognised by performance in competitions; these competitions tend to reward the best all-rounders.

Dang [76] argues that complementarity is important and therefore a good addition to a portfolio is not necessarily the best all-round performer, but rather the algorithm which fills in the gaps left by the existing members of the portfolio.

3.3.2 Measuring Performance

Before referring to some relevant work on algorithm selection, it may be beneficial to discuss how we might measure the effectiveness of a solution – in our case a solution which brings to bear ML-based predictions to solving constraint problems. In the previous chapter I have already mentioned competitions for solving CSPs and SAT. Each of those competitions has their own scoring mechanisms, but there tend to be some common components.

What to Measure

Here are various metrics which can contribute to a final score or judgement, be it for a solving competition, or simply for judging the effectiveness of a proposed new system.

penalised average runtime (PAR) The average running time is calculated for a solver over a set bechmark of problems. If a problem is not solved within a set timeout period, then that run is given the value of $p \times t$ where p is a penalty factor, and t is the timeout period. A tenfold penalty is very common, and the corresponding figure is called PAR10. PAR5, PAR2 and PAR1 are also found in literature. A reasonable penalty factor takes into account the distribution of solving times. For example, if it is possible that a solution will never be found, then a higher penalty should be applied to timed out runs. In PAR the arithmetic mean is most common, but other averages could also be used.

# of Instances Solved We can simply set a timeout and report how many of the instances in the problem corpus were individually solved within that time. Some care may be needed if the solving strategy involves randomness. It may be fairer to make several runs of each solver on each problem and use an average. The number of instances solved can also be visualised as a "cactus" or "survival" graph to show how the timelimit interacts with the number of problems solved.[2] An example is shown in Figure 3.2, comparing 19 years' worth of SAT competition winners.

Speedups Another way to report the performance of a proposed solution is to calculate the speed up achieved compared to the state of the art, or some other clear baseline. When using speedup ratio as a measure, one must be careful and clear about how this

[2] Although "cactus" and "survival" are sometimes mixed up, the cactus plot should have # of instances along the x axis, with the plots shooting up like a cactus and better solutions being further right, whereas a survival plot has the time on the x axis, so higher is better.

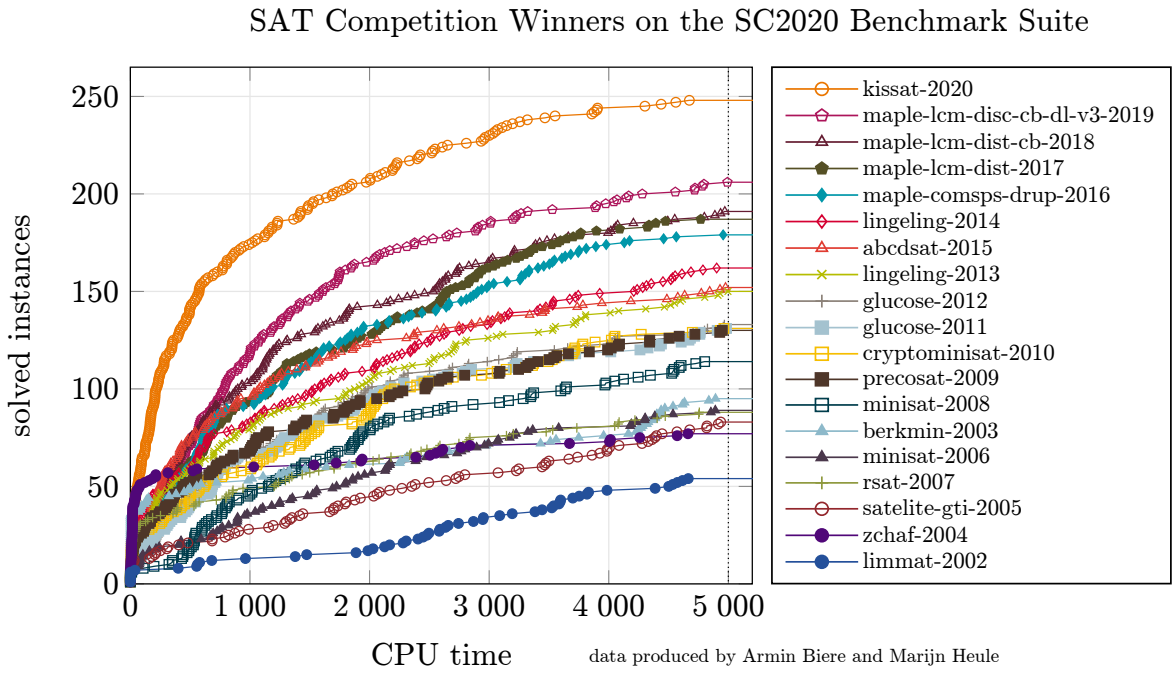


Figure 3.2: An example of solver comparison using a survoval plot: instances from the 2020 SAT Competition solved by historical winning solvers. Plot from http://fmv.jku.at/kissat/

is aggregated to arrive at a final figure. If we had a thousand fold speed up on an easy problem but took twice as long on a harder problem it would be grossly misleading to report an average speed up of $(1000 + 0.5) \div 2 = 500.25$.

Pseudo-harmonic Mean One example of a novel measure used in a particular context is the *pseudo-harmonic average distance from the solutions found and the best solutions known* introduced in [50]. The authors calculate for any solution given a distance from the best-known solution. Because sometimes no solution is reached within the timeout, this distance ends up being infinite, and their custom metric deals with this to give a meaningful and interpretable measure.

Offline Time and Energy

A potentially important consideration when comparing systems is the amount of time taken to train the model – this time is generally not taken into account when comapring the solving performance of an ML-enhanced CO algorithm. At runtime the ML model can usually provide predictions almost instantly, but in the offline phase, many CPU years may have gone into training the model.

In our age of ML algorithms which train on billions of data points and use a tremendous amount of computing power (and energy), one could argue that a balance needs to

be struck between sheer performance and the energy expended to secure that speedup. A full discussion of green AI issues is beyond the scope of this thesis (see [77] for a review of green AI research), but it's an issue worth bearing in mind and it may be preferable to favour a solution which is simpler (or less resource-hungry) in training if it is still competitive according to the relevant metrics.

3.3.3 Algorithm Selection Case Studies

Kotthoff's survey of algorithm selection approaches [78] explains that the Algorithm Selection Problem is frequently applied in Artificial Intelligence contexts. In this paper many aspects and stages of this paradigm are explored specifically in the field of "Combinatorial Search" – essentially CSPs. For example, some systems use a rank-based scheduling approach, i.e. they predict the relative expected performance of the various algorithms for the problem in question and then they allocate CPU time in proportions according to the ranking. Other systems start a number of algorithms in parallel, then perform some analysis of progress and then re-schedule CPU time accordingly.

Below I present some existing solutions which apply portfolio approaches to some aspect of solving constraint problems.

SATzilla

One celebrated example of algorithm selection is the SATzilla portfolio SAT solver [79]; this solver won several medals in the 2007 and 2009 SAT competitions. It chooses which specific solver to call based on extracting features of the SAT problem instance. SATzilla also includes a "pre-solve" stage where it runs a set of pre-solvers for a fixed time – the extracted metrics then predict the best solver to choose to solve the problem.

As the first portfolio-based SAT solver to achieve success in international SAT solving competitions, SATzilla made a significant leap forward in performance. Their approach, according to the authors, "can be considered to be a classifier with an error metric that depends on the difference in runtime between algorithms." One interesting distinction they make with other work is that they do not equally penalise the not-best-performing solvers, rather recognising that some solvers are nearly the best. This enables them to sometimes choose a non-optimal (but nearly-optimal) solver quickly and get on with solving the problem, achieving a better solving time overall.

The performance gain from applying a portfolio was so great, that the SAT competitions eventually banned portfolio approaches in order to encourage developers to focus on

improving the SAT solving algorithms themselves.

SUNNY-CP

SUNNY-CP was a very successful portfolio-based constraint solver, which was adapted from just working on CSPs to deal with COPs too in [80]. It labels itself as a "*lazy* portfolio-based approach". It uses a portfolio of 8 solvers: CPX, G12/FD, G12/LazyFD, G12/CBC, Gecode, MinisatID, Chuffed, and G12/Gurobi. SUNNY-CP operates by extracting instance features from MiniZinc problem descriptions. It then consults either a CSP-oriented knowledge base, or one trained on COPs, based on the problem at hand. An algorithm based on KNN regression is then applied to construct a schedule of solvers to run.

MeSAT

Tackling a challenge closely related to the theme of this thesis, the authors of MeSAT [81] present an ML-based system for choosing between SAT encodings for CSPs. They propose combinations of existing SAT encodings for integer variables – a KNN algorithm is then used to select between the menu of encoding choices based on features of the problem instance at hand. The authors claim that it is possible to learn to make good choices from easier problems – the trained model can then extrapolate to harder instances and still make good predictions which lead to better performance than using the best single encoding.

Variable Encodings MeSAT uses the existing *direct*, *support* and *log* encodings for integers, as well as creating some hybrid encodings by using clauses from two base encodings to create the *direct-support* and *direct-order* encodings.

Constraints The authors adapt the structure of existing SAT encodings for global constraints to make use of the hybrid variable encodings they proposed. Global constraints affected include *all-different*, *global-cardinality*, *nvalue* and *cumulative*.

ML Setup MeSAT uses syntactic features of the constraint problems for its learning and prediction. These 70 features consider the distribution of the types of constraint in a problem, as well as the types of variables and their domains. The chosen model is K-Nearest Neighbours, where a new instance is assigned the same SAT encoding scheme that worked best for the nearest k training instance in the feature space, when considering PAR10 solving time.

Performance The experiments showed that the portfolio approach improved the number of instances solved compared to picking any single encoding scheme. It was a little difficult to assess the performance fully, as the penalty given for a timed out run was the timeout time (i.e. PAR1). When the authors tried to speed up training by only training on easy instances, they still achieved some success, but much more limited than when training on a uniform sample of the instances available in their corpora.

PROTEUS

Operating in a similar space, but at a higher level of abstraction, the PROTEUS "hierarchical" portfolio solver [82] takes a CSP (or COP) as input and makes decisions in three stages; first it considers whether to use a specialised constraint solver or to translate the problem to SAT. In the second case it then chooses which SAT solver to employ and finally selects between different encodings of the problem to SAT.

Solvers For its constraint solvers, PROTEUS can choose between Abscon, Choco, Mistral and Gecode. If a problem is encoded to SAT, the solvers available are: Clasp, Cryptominisat, Glucose, Lingeling, MiniSAT and Riss. Although the choice of solvers is not specifically justified in [82], it is likely to have been a representation of the state of the art at the time, with solvers having complementary strengths.

Encodings The *direct* and *support* encodings feature as they did in MeSAT. In PROTEUS there is also a hybrid *direct-order* encoding. Although global constraints are mentioned, it is not clear exactly which global constraints are implemented and how they are translated to SAT.

ML models The authors describe trying out many ML algorithms; their chosen setup is to use regression models (mostly linear regression) to make the decisions at the three levels.

Performance PAR10 is used to compare the performance of PROTEUS with the best possible performance achievable by using any of the single routes, such as solving them all by a direct constraint solver. For example, the results show a twofold improvement compared to always going directly to CP solvers, but being able to chose the best solver for the instance (the authors call this VB CSP for virtual best using CSP solvers), and a tenfold improvement compared to always using SAT. This could be interpreted as discouraging

to the SAT route, but the paper itself shows that their problems are roughly evenly split between ones that do better with SAT solvers versus CSP solvers. Furthermore, in more recent competitions, SAT-based constraint solvers have consistently performed best and work has continued on SAT encodings for specific constraints which greatly improve performance in certain settings.

AutoFolio

In contrast with the other systems described above, AutoFolio [83] is a general algorithm selection tool which the authors are confident "can be applied out-of-the-box to previously unseen algorithm selection scenarios". In the empirical evaluation, the authors do compare AutoFolio's performance with other algorithm selection solutions on benchmarks comprised of CSP, SAT and related problems and show impressive performance wins over systems which are more specialised to the particular class of problems solved. AutoFolio is a very sophisticated system which potentially does much more than our context requires. For instance it is able to produce a schedule of solvers to try, in a sense placing bets on which algorithms might pay off very quickly.

AutoFolio essentially operates by considering a set of existing ML-based algorithm selection solutions which have been pre-tuned. AutoFolio re-configures the hyperparameters for these selectors according to its own training data by using a technique called sequential model-based algorithm configuration (SMAC). By estimating the likely runtime, it then creates a schedule of solvers to try. In its implementation AutoFolio uses several ML techniques introduced above, including random forests, gradient boosted trees, and k-means clustering.

3.4 Summary

In this chapter I have introduced some key concepts in the area of machine learning (ML), and explored some research which has used ML techniques to improve the performance when it comes to solving constraint satisfaction and optimisation problems.

I gave an overview of the broad categories of machine learning along with examples of how they might be applied and some of the advantages and limitations of various ML algorithms. I attempted to lay out some of the important issues relevant to the design of a system based on ML. We saw that a lot of work has been done in bringing together ML and constraint solving (sometimes referred to as CO or combinatorial optimisation). The

way ML and CO interface can be very different with work ranging from setting some initial parameters for CO algorithms using ML predictions to ML systems which aim to carry out the entire solving process.

Many choices exist at every stage of the solving process, all of which can be delegated to a trained ML "assistant". We looked at how portfolios of algorithms can be constructed and chosen from to perform sub-tasks in the solving process.

We finished by summarising some work which specifically related to choosing SAT encodings for aspects of a CSP by employing ML methods.

In the next chapter, I present my first attempt to choose SAT encodings using ML for a specific type of constraint.

Chapter 4

Learning to Select Encodings Using Generic Instance Features

♫ You're simply the best ♫

Tina Turner

Research Question 1

To what extent does the choice of SAT encoding for constraints affect problem-solving performance?

Research Question 2

Can a good SAT encoding choice be made for unseen CSP instances based on generic instance features?

We saw in Chapter 2 that encoding CSPs to SAT can be a very effective way to solve them, but that the encoding scheme chosen for variables and constraints can make a big difference to the solving performance. In Chapter 3 we reviewed work which uses machine learning to make good choices about how to encode various aspects of a CSP into SAT – choices leading to faster solving times on average. These works principally focussed on the encoding chosen for the CSP variables. The focus of my thesis is on selecting encodings for the constraints themselves.

The previous related works (MeSAT and Proteus) contain instances of the same problem class in their training and test sets, in other words they make predictions for instances of problems which have already been seen in the training phase. In this chapter I attempt

a further scenario where predictions are made for instances of problem classes which have not been seen during training. This is both a much more challenging proposition and provides the basis for making a good initial decision for a user who hasn't had the time or expertise to fine-tune the encodings for the problem they're attempting to solve.

In this chapter I describe my first attempt to select good SAT encodings for linear integer (LI) and pseudo-Boolean (PB) constraints. PB and LI constraints are covered extensively in literature and have several SAT encodings available. Empirical studies have shown that different encodings are suited to different problems – the expectation is that it should be possible to predict which encoding would be a good choice for a problem instance based on its features. For the sake of readability I will often refer to the process outlined in this chapter as LeaSE-PI, which stands for **Lea**rning to **S**elect **E**ncodings **P**er **I**nstance.

The chapter contains:

- an outline of the stages and design decisions involved in conducting an empirical investigation into the research question posed above
- a detailed description of the experimental setup
- a presentation of the results of the experiment
- a discussion of the findings

4.1 Method

This initial investigation considers a diverse collection of problems and goes on to extract features of the problem instances, train some classifiers which then make predictions on held-out sets of problems. These predictions are then evaluated to address the research question posed above.

4.1.1 The problem corpus

In order to have access to a wide variety of problem classes, I begin with a corpus from a recent paper [51] which uses Savile Row. The collection of 65 constraint models with a total of 757 instances has the added advantage that the models are written in Essence Prime, Savile Row's input language. I also add two problem classes from recent XCSP3 competitions [84]: the Balanced Academic Curriculum Problem (BACP) and the Hamiltonian Cycle Problem (HCP) (the XCSP3 language and competitions were introduced in Section 2.1.5).

Here is a brief overview of just some of the problem classes and the categories they belong to into to give the reader an idea of the variety in the corpus:

Combinatorial Problems

Balanced Incomplete Block Design A classic combinatorial problem of arranging objects into blocks such that each object appears a given number of times and pairs of objects appear together a set number of times

Langford's Arranging pairs of blocks so that blocks of the same colour have a distinct and contiguous number of blocks between them

Equidistant Frequency Permutation Arrays Finding a set of codewords so that any pair has a given Hamming distance

Scheduling

Nurse Rostering Creating a weekly schedule respecting working rules and nurses' preferences

Balanced Academic Curriculum Assigning students to courses, with required credits per semester and course pre-requisites

Car Sequencing Ordering cars to enable different options to be installed according to assembly-line restrictions

Games and Puzzles

N Queens Placing queens on a chessboard so they don't attack each other

Blackhole Solitaire A patience card game requiring steps to potential completion

Sudoku A grid puzzle where the numbers in each row, column and "box" have to be distinct

Unfortunately this collection has a very skewed distribution of instances per problem class, ranging from just 1 to 100. To mitigate this, I firstly limit the number of instances per class to 50 by taking a random sample where more instances are available; secondly, I add instances to existing classes where it is easy, such as when instance parameters are just a few integers. Some of the problem instances and indeed whole problem classes end up being dropped in the cleaning phase of dataset preparation, which is described in Section 4.2.2.

4.1.2 Features

Using FZN2FEAT to obtain the *f2f* featureset

We saw in Chapter 3 that features of CSP instances can be extracted and used to make predictions at various points in the solving process. I begin by using the static features extracted by the FZN2FEAT tool, created by Amadini et al [85] as part of the higher-level feature extractor MZN2FEAT. These 95 instance features relate to the number and types of variables and their domains, the types and sizes of constraints and features of the objective in optimisation problems. From now on we will call this featureset *f2f*. The full list of features can be found at `https://github.com/CP-Unibo/mzn2feat` and is reproduced in Appendix A.

The FZN2FEAT tool requires `FlatZinc` models as input – we generate these using SAVILE ROW's standard `FlatZinc` back-end. Some features are not applicable, e.g. there are no float variables in Essence Prime and SAVILE ROW does not produce all the same annotations.

Using SAVILE ROW's internal model to generate the *f2fsr* features

In our context the ultimate aim is not to produce `FlatZinc`; our intended output is a SAT formula. SAVILE ROW applies various reformulations with the target solver in mind, so it may be preferable to consider the constraint model targetting the SAT back-end just before encoding to SAT takes place, rather than targeting `FlatZinc` simply to produce instance features, and then running SAVILE ROW again to actually solve the problem using a SAT back-end.

I reproduce the features in *f2f* as faithfully as possible based on the internal model in SAVILE ROW just before the final step of producing the SAT formula. I call this featureset *f2fsr*. As well as representing a more relevant constraint model, the use of this featureset has the advantage that it is quicker to generate because SAVILE ROW does not need to be run twice.

4.1.3 The Encodings

The PB (and therefore LI) encodings available in SAVILE ROW include all AMO-aware encodings introduced in Section 2.3.2. Here is a brief reminder of their design.

MDD binary decision trees

GSWC counter logic circuit

GGT binary "totalizer" tree with all possible sums represented in node variables, here built with the *minRatio* heuristic

GGTd as above, but using a balanced tree

RGGT another totalizer, this time using intervals where possible to reduce the number of variables

GMTO a mixed-radix totalizer

GGPW uses a bitwise formula to spot constraint violation

GLPW bitwise arithmetic enforcing arc consistency for each variable

In addition, recall that Savile Row also provides the *Tree* PB and LI encoding which is not AMO-aware. *Tree* does UP-maintain GAC, except when the comparator is $=$ or $\neq$.

4.1.4 Training

Classifiers

I have evaluated several classifier models from the `scikit-learn` library [6], including forests of extremely randomised trees, K-nearest neighbours, multi-layer perceptrons, as well as the XGBoost classifier [63]. In Appendix B I include an example of one of the exploratory experiments trying out a neural network classifier and comparing it to the performance of the setups fully featured in this dissertation.

While searching for the best prediction model I have also experimented with regressors. I have trained regressors to predict the runtime for the different encoding configurations in order to select the encoding with the lowest predicted runtime. For each ML model tested (decision trees, random forest, gradient boosting), the regressor approach performs worse than using the classifier variant.

After exploratory experiments I conclude that three classifiers work particularly well for our task: decision trees, random forests and gradient boosted decision trees.

For the decision tree classifier, I tune the *splitting citerion*, *maximum tree depth*, the *maximum number of features* and the *minimum number of samples per leaf*.

The `scikit-learn` implementation of random forests is based on Breiman's work [60], but uses an average of predicted probabilities from its decision trees rather than a simple

vote. I implement the recommendations of Probst et al. [86] 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*.

For the gradient boosted trees I use the histogram-based gradient boosting classifier from `scikit-learn` which is "inspired by LightGDM" [87]. Gradient boosting creates an ensemble of trees sequentially, at each stage putting more emphasis on the training examples which have been most badly misclassified so far. I tune the *learning rate*, *maximum tree depth* and *maximum number of iterations*.

I use randomised search with 50 iterations and 5-fold cross-validation to tune the hyperparameters for all classifiers mentioned above. I experimented with more tuning iterations but it did not lead to improved prediction quality.

A Pairwise Approach with a Complementary Portfolio

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, I follow a similar approach to the *pairwise classification* used in AutoFolio [83]: an ML model is trained for each of the possible pairs of encoding configurations. An encoding configuration is a (*LI encoding*,*PB encoding*) pair drawn from the encodings summarised in Section 4.1.3.

When making predictions, each model chooses between its two candidate configurations. The configuration with most votes is chosen; if two or more configurations have equal votes, the winner is 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 often leads to better performance than using a single random forest classifier. Taken as a whole, the pairwise setup has access to richer training data in the sense that instead of each instance having one label with the target encoding, we have a verdict on each pairwise contest. The voting mechanism is also a good guard against constly misclassification – a bad choice of encoding would have to win several contests to become the eventual winner.

Given that 81 encoding configurations are possible (9 PB encodings $\times$ 9 LI encodings) we would need to tune and train $\binom{81}{2} = 3240$ classifiers, which is computationally expensive. Fortunately it is not necessary - it is possible to approach the virtual best perfor-

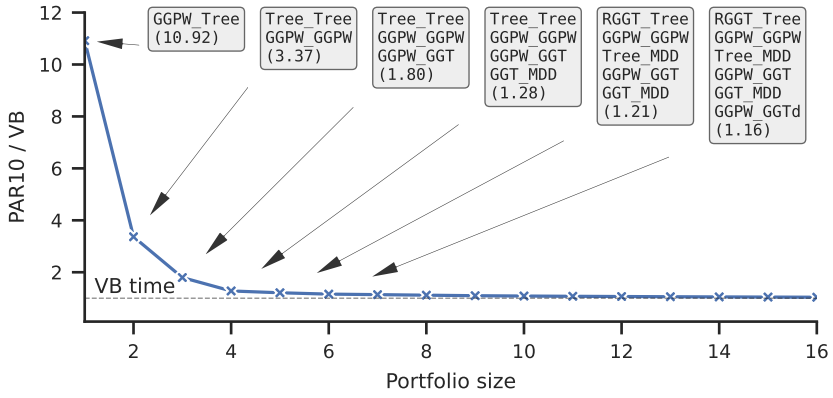


Figure 4.1: The virtual best PAR10 run-time on our whole corpus for a range of portfolio sizes, as a multiple of the overall virtual best; the resulting portfolios (of *li_pb* configurations) are shown for sizes 1 to 6.

mance even if we limit ourselves to a subset of the configurations available by building a suitable portfolio.

I seek to retain performance complementarity as described in [88] from a much reduced portfolio size. The portfolio is built from the timings in the training set using the *sequential forward* method described in [74]. It is a greedy approach and works as follows. I begin with a single encoding configuration in the portfolio and then successively add the configuration which would lower the virtual best PAR10 time by the biggest margin. Configurations are added in this manner until we have a portfolio of 6. I repeat the process using each of the 81 configurations as the starting element and finally select the best-performing portfolio from these 81.

To investigate how close to the virtual best we can get when we reduce the number of configurations available, I ran the portfolio building algorithm on the entire corpus of problems. Figure 4.1 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 6 is within 16% of the time achievable with all configurations. As fas as I'm aware this is a novel application of a portfolio approach to the selection of SAT encodings.

In addition to the pairwise voting scheme, I implement two further customisations when training the classifiers:

Sample Weights Firstly we aim to give more importance to instances which are harder (with a longer virtual best runtime) and where the encoding choice makes a bigger difference. Each of these two criteria would make misclassification potentially more costly, that is, in contrast to a problem which was either quick to solve or where

all encodings performed similarly. To implement this prioritisation, each instance is given a positive integer weight w according to the formula

$$w = \lfloor \log_{10}(10 + t_{VB} \times \frac{t_{VW}}{t_{VB}}) \rfloor = \lfloor \log_{10}(10 + t_{VW}) \rfloor$$

where t_{VB} and t_{VW} are the very best and very worst runtimes respectively for the instance. [1] The *hardness* is represented by t_{VB} and the *variability* by $\frac{t_{VW}}{t_{VB}}$. The product of these two components cancels to just t_{VW}, so the resulting weighting is essentially the order of magnitude of the very worst time.

Custom Loss Secondly, I provide a custom loss function for the cross-validation used during hyperparameter tuning. This function simply returns the difference in time between the runtime of the chosen encoding configuration and the virtual best for that instance.

In order to provide a more complete comparison I also implement two additional alternative setups:

Single Classifier I use a single classifier with the same portfolio of 6 configurations (combining PB and LI encodings). I use a generous allowance of 15 times the iterations allocated to each classifier in the pairwise setup to compensate for the fact that the single classifier is doing the work of the 15 pairwise classifiers. I also include another setup with a more restricted budget of just 60 iterations.

Separate LI/PB Choice I modify the pairwise setup to make a separate prediction for LI and PB constraints, choosing a portfolio of 6 encodings for each encoding type. This approach has its difficulties because when labelling the dataset with the best encoding for one type of constraint, the encoding of the other constraint type must be chosen somehow – I set this to the single best for the training set. This setup is more expensive in terms of training time, effectively repeating the entire process for each constraint type under consideration, rather than taking advantage of a complementary portfolio across two (or more) encodings. However, if one expands LeaSE-PI to include other constraint types beyond PB and LI, this approach would scale better, given that each extra constraint type introduces another dimension in the configuration space.

[1] The weighting does not need to be an integer as far as the classification algorithms are concerned, but the initial intention was to let us group problems for the sake of discussion.

4.2 Experimental Setup

The experimental process is illustrated in Figure 4.2 and it involves:

- Running Savile Row with different encoding choices in order to collect runtime information and to extract features.
- Cleaning the resulting dataset.
- Carrying out 50 *split, train, predict* cycles with each of our machine learning setups, using the same train/test splits in order to allow fair comparison across the setups.
- Using the predicted encoding choices to identify the resulting runtimes.
- Aggregating the "predicted" runtimes and calculating reference times for comparison.

In this section I set out the details of how the experiment was designed and conducted.

4.2.1 Solving Problem Instances and Extracting Features

I run Savile Row on every instance in the corpus with each of the 81 encoding configurations. The CNF clause limit is set to 5 million and the Savile Row time-out to 1 hour. Automatic detection of At-Most-One constraints [52] is switched on. I use Kissat as the SAT solver because it formed the basis of the top performers in the 2021 and 2022 SAT competitions [32]. I use the latest release available at the time, `sc2021-sweep` [89], with default settings and a separate time limit of 1 hour. The experiment is run on the Viking research cluster with Intel Xeon 6138 20-core 2.0 GHz processors; the memory limit for each job is set to 8 GB. I carry out 5 runs (with distinct random seeds) for each configuration to average out stochastic behaviour of the solver.

To extract the features I run each problem instance once with the Savile Row feature extractor and once to generate a standard FlatZinc file (using the `-flatzinc` flag) followed by `fzn2feat` [85]. The time required to extract the features is also recorded.

4.2.2 Cleaning the Dataset

I calculate the median runtime over 5 runs for each instance and encoding configuration. A result is marked as timed out if the total runtime (Savile Row+ Kissat) exceeds 1 hour. To decide what penalty to apply to runs which time out, I consider all instances for which

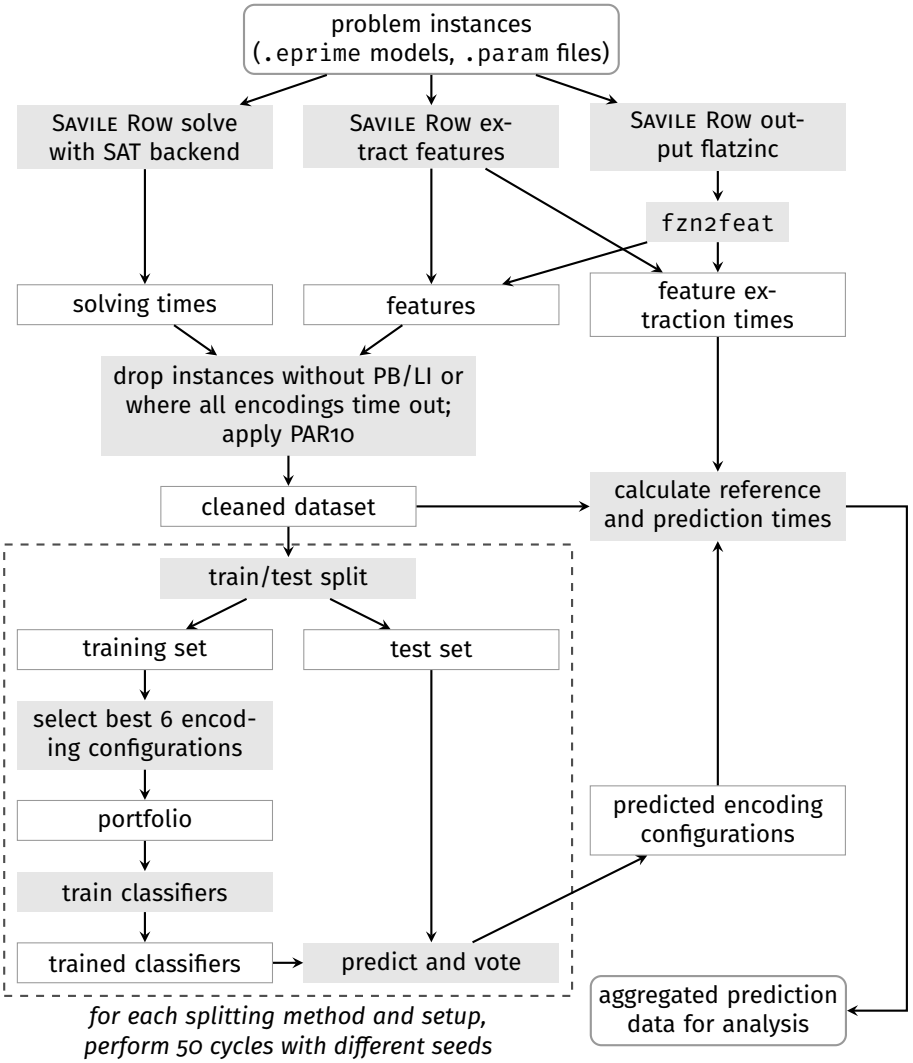


Figure 4.2: An overview of the steps involved in our experimental investigation. The white boxes with solid borders represent data; the grey boxes represent processes.

every configuration finishes within the allocated time. The mean *worst*/*best* ratio is 13.06 and the median ratio is 4.91. I also consider those problem instances which are not solved with any encoding configuration in less than 1 minute, i.e. harder problems. In this case the *worst*/*best* mean ratio is 9.18 so I believe it fair to penalise a time-out by a factor of 10. I therefore use PAR10, i.e. assigning 10 hours to any result which takes longer than our 1 hour time-out limit. This is the same penalty applied in other related literature [82, 83, 81] which addresses the problem of selecting SAT encodings for CSPs.

Having applied PAR10, I filter the corpus as follows. Instances are dropped if they contain no PB or LI constraints. I also exclude any 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. Finally I exclude instances for which all configurations time out. At this point, 614 instances of 49 problem classes remain in the

Table 4.1: Number of instances *(#)*, mean number of PB constraints *(PBs)* and mean number of LI 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	36	21.9	76.2	knapsack	28	1.0	1.0
sonet2	24	10.0	1.0	immigration	23	0.0	1.0
bibd-implied	22	410.6	0.0	efpa	21	162.8	0.0
handball7	20	705.0	1206.0	mrcpsp-pb	20	90.0	45.7
n_queens	20	1593.0	0.0	bibd	19	338.7	0.0
briansBrain	16	0.0	1.0	life	16	0.0	438.9
molnars	16	0.0	4.0	n_queens2	16	309.0	0.0
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	magicSquare	7	136.0	36.0
peaceArmyQueens1	7	0.0	1008.0	peaceArmyQueens3	6	0.0	4.0
quasiGrp5Idem	6	586.7	0.0	golomb	6	59.2	38.7
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				

corpus; Table 4.1 shows the number of instances for each problem class and the mean number of PB and LI constraints per instance.

4.2.3 Splitting the Corpus, Training and Predicting

For each of the classifier setups and featuresets, I run a *split, train, predict* cycle 50 times. Random seeds 1 to 50 are used to co-ordinate the splits so that I can compare the prediction power of the different feature sets and setups using the same training and test sets.

For each cycle, the aim is an 80:20 *train:test* split using two approaches. The *split-by-instance* approach simply selects instances at random with uniform probability – with this sampling method, instances of any given 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 of a problem class end up either in the training or the test set, meaning that predictions are being made on unseen problem classes. With this method, a minimum proportion of 20% is enforced for the test set. Due to the varying number of

instances per problem class, the test set can end up being slightly larger than 20% and correspondingly the training set is then slightly less than 80%.

Prior to training the classifiers, the portfolio of available configurations is built based on the runtimes of the training set. Then the training instances are labelled for each pairwise classifier with the configuration that has the shorter runtime. For each pairwise classifier, the hyperparameter space is randomly searched and the model is fit to the training set. Finally, the models are used to make predictions using the test set ready for evaluation.

4.2.4 Evaluating Performance

To evaluate the impact of using the learnt encoding choices, I calculate two benchmarks commonly used in algorithm selection [88]: the *Virtual Best (VB)* time is the total time taken to solve the instances in a test set if we always made the best possible choice from all 81 configurations; and the *Single Best (SB)* time is the total time taken to solve the instances in a test set when using the single configuration that minimises the total solving time on the training set. In addition we consult: 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. From here onwards I sometimes refer to these selection methods as *selectors* along with the ML models which select encodings based on sets of features.

4.3 Evaluation

4.3.1 Results

In Table 4.2 we see the total PAR10 runtime across all fifty test sets for the predicted encoding configurations from each of the eleven classifier setups, four feature sets and two splitting methods. The predicted runtimes include the time taken to extract the features.[2] For ease of comparison, I give the runtime as a multiple of the virtual best time. For example, a figure of 2.00 in Table 4.2 means that the predictions across the fifty test sets led to a total runtime which was twice as long as the runtime achieved if we always chose the best

[2] For the *f2fsr* (extracted directly from Savile Row), the feature extraction time added a median of 9% (mean 21%) to the overall running time. The features extracted via `fzn2feat` added 66% (median), 72% (mean).

available configuration. The reader is reminded that the two splitting strategies (by class vs. by instance) yield different test sets for the fifty seeds, as explained in Section 4.2.3.

The machine learning predictors work well, outperforming the *SB* and *Def* configurations. In the majority of cases the *f2fsr* features (extracted directly from Savile Row) lead to better predictions than the ones resulting from exporting to FlatZinc first (the *f2f* featureset). It is difficult to declare a particular setup as the best overall – the gradient boosting algorithm works best for the *split-by-instance* setting, the decision tree performs well for *split-by-class*. For the classifiers based on random forests, the sample weighting and custom loss functions do improve performance.

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" [88]. In these terms, the ML-based predictions close up to 46% of the VB-SB gap for unseen classes and 81% for seen classes. The *split-by-class* setting is both a more difficult challenge and closer to a real-world deployment, where a user may seek to solve a new problem on whose class the ML model has not been previously trained. However, both settings have realistic applications. To carry out further analysis I therefore select a "preferred" setup which performs competitively in both settings (by instance and by class). I choose the fourth entry in the table: pairwise random forest classifiers with sample weighting and custom loss, using a portfolio of combined configurations for PB and LI encodings. This setup shows less variability across the featuresets and still closes up to 71% and 41% of the VB-SB gap for seen and unseen problam classes respectively.

Figure 4.3 summarises the performance for our preferred setup. On the left it shows the distribution of mean predicted PAR10 times per test set. The mean values are marked with diamonds and correspond to the numbers reported in Table 4.2, albeit not scaled. On the right I report the distributions of timeouts per test set for each selector. Recall that we cleaned our corpus so that each remaining instance could be solved using at least one encoding configuration – this explains why there are no timeouts with the VB selector.

We see that the runtimes are markedly more skewed when splitting by class, indicating that in one or more of the 50 trials, the choice made by all four non-VB selectors was rather poor, leading to some combination of longer runtimes and more timeouts. This effect is not observed when splitting by instance because the instances of any problem class are divided between the training and test set (in a ratio of 80:20), meaning firstly that any challenging problem classes are only contributing a small number of instances to the test set, and secondly that when it comes to the ML classifiers and the SB classifier,

Table 4.2: Total PAR10 times over the 50 test sets as a multiple of the virtual best configuration time. The best time for each combination of setup and splitting method is shown in **bold**. The predicted runtimes include feature extraction time. In the setup details, the classifiers are random forest (*RF*), decision tree (*DT*) or gradient boosted (*GB*) and they either predict for *pairwise* combinations of encodings or make a *single* multi-label prediction; *Co/Sep* shows whether LI and PB encodings were selected separately or as a combined choice; *SW* means sample weighting is used; *CL* indicates custom loss used in cross-validation; *Tuning* refers to the number of cycles of hyperparameter tuning.

Split-by-Instance Reference Times

Virtual Best	Single Best	Default	Virtual Worst
1.00	13.87	17.84	123.70

Split-by-Instance Predicted Times

Setup Details						Features	
Classifier	Selector	Co/Sep	SW	CL	Tuning	f2f	f2fsr
RF	Pairwise Voting	co	-	-	50×15	7.99	**5.63**
RF	Pairwise Voting	co	✓	-	50×15	6.33	**5.36**
RF	Pairwise Voting	co	-	✓	50×15	6.01	**4.74**
RF	Pairwise Voting	co	✓	✓	50×15	5.40	**4.69**
RF	Single Classifier	co	✓	✓	750	3.91	**3.70**
RF	Single Classifier	co	✓	✓	60	3.95	**3.70**
RF	Single Classifier	co	-	-	750	10.34	**9.64**
RF	Pairwise Voting	sep	✓	✓	$50 \times 15 \times 2$	**6.53**	6.67
DT	Pairwise Voting	co	✓	✓	50×15	**6.00**	6.06
GB	Pairwise Voting	co	✓	✓	50×15	4.25	**3.97**
GB	Single Classifier	co	✓	✓	750	4.25	**3.44**

Split-by-Class Reference Times

Virtual Best	Single Best	Default	Virtual Worst
1.00	25.40	17.15	160.96

Split-by-Class Predicted Times

Setup Details						Features	
Classifier	Selector	Co/Sep	SW	CL	Tuning	f2f	f2fsr
RF	Pairwise Voting	co	-	-	50×15	15.07	**15.01**
RF	Pairwise Voting	co	✓	-	50×15	16.80	**14.90**
RF	Pairwise Voting	co	-	✓	50×15	16.42	**15.30**
RF	Pairwise Voting	co	✓	✓	50×15	15.69	**15.19**
RF	Single Classifier	co	✓	✓	750	20.59	**19.15**
RF	Single Classifier	co	✓	✓	60	22.01	**19.49**
RF	Single Classifier	co	-	-	750	**19.72**	19.93
RF	Pairwise Voting	sep	✓	✓	$50 \times 15 \times 2$	16.91	**14.10**
DT	Pairwise Voting	co	✓	✓	50×15	17.63	**14.05**
GB	Pairwise Voting	co	✓	✓	50×15	19.26	**17.45**
GB	Single Classifier	co	✓	✓	750	24.54	**19.76**

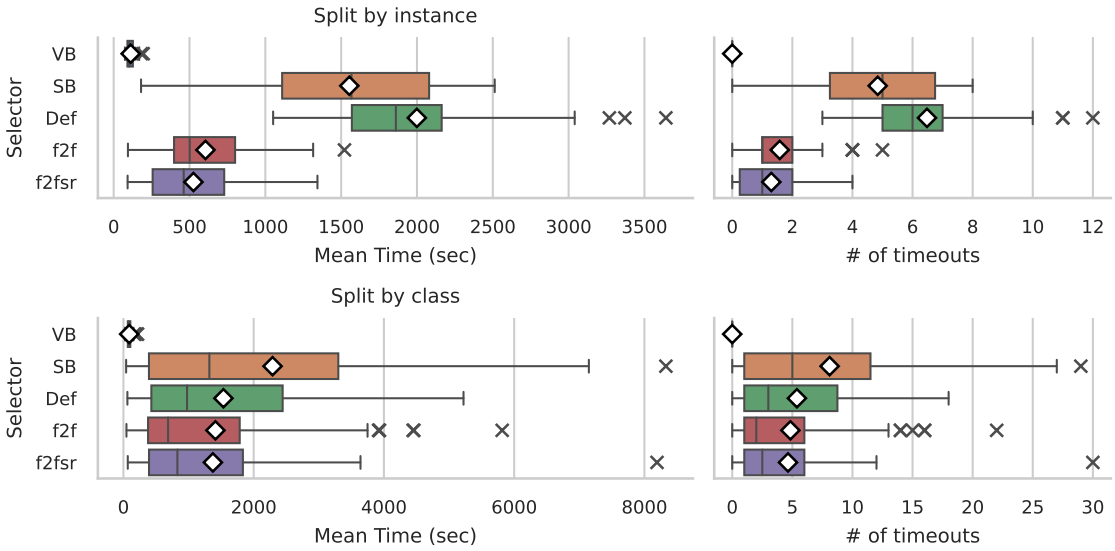


Figure 4.3: Prediction performance using *f2f* and *f2fsr* features against reference times. We show mean PAR10 runtime (left) and number of timeouts (right) per test set across the 50 cycles with our preferred setup (*pairwise random forests, sample weights, custom loss*). Outliers are indicated with crosses and represent values more that 1.5 × IQR outside the quartiles.

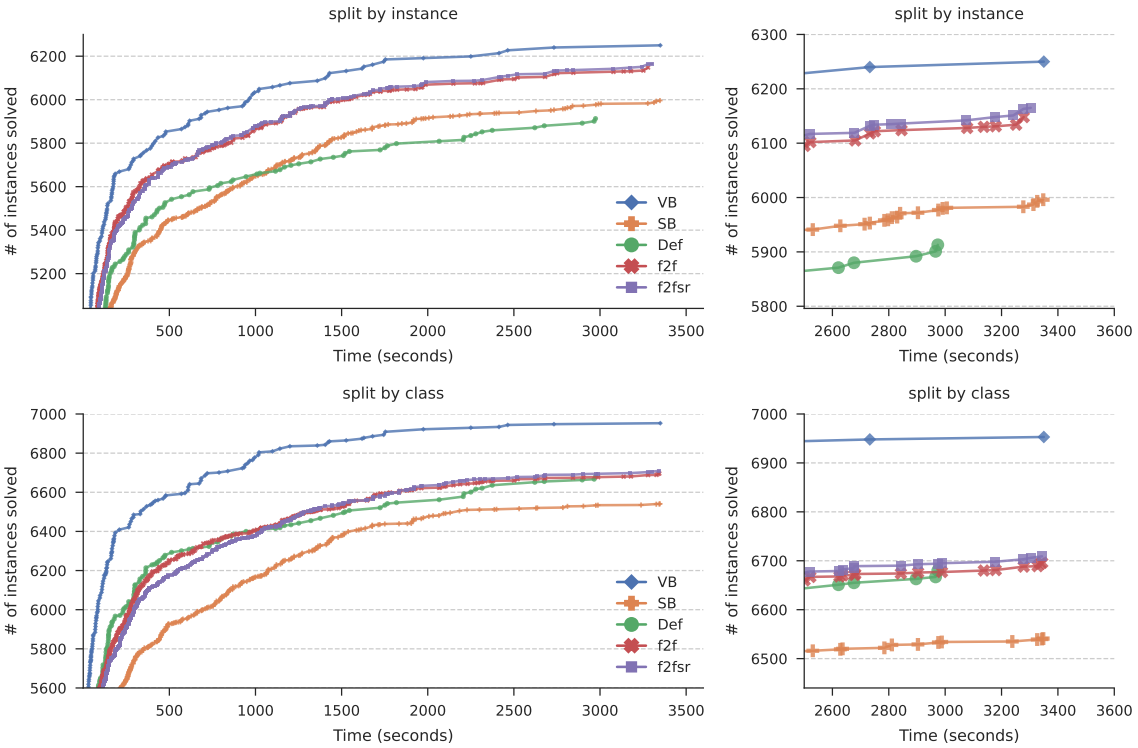


Figure 4.4: The number of problem instances individually solved within a given time for the reference selectors and our preferred predictor using different feature sets. The figures on the left show the full performance profile; on the right we zoom in to see how many instances are solved by the selectors as we approach our timeout limit of 1 hour.

each problem class has been considered in the training phase.

In terms of timeouts the ML-based choices are better than both the single best and default choices, although in the *split-by-class* setting this is only narrowly the case. When predicting encodings for seen problem classes, the ML models are able to bring down the mean number of timeouts per test set dramatically from 5 and 6 for *SB* and *Def* to below 2. In the *split-by-class* setting, this reduction is more modest, bringing the mean down to just below 5 from just above 5.

One striking observation is that in the case of unseen problem classes one particularly bad trial has dragged up the mean runtime, taking away from what would otherwise have been a much better performance for the *f2fsr* features. We observe this in both the number of timeouts and the PAR10 plots. The lowest box plots show that in one trial there were 30 timeouts (out of approximately 130 instances). With the tenfold penalty, this trial has a mean PAR10 time of over 8000 seconds, compared to the next worst trial, which has a PAR10 mean time of less than 4000 seconds.

An unexpected aspect of the results is the relative performance of the default (*Def*) and single best (*SB*) configurations. When splitting by instance, the result is not surprising, with the SB outperforming the default. However, this is reversed when splitting by class. This indicates that in some cases the best encoding choice for the problem classes in the training set performs extremely badly on the instances in the test set. This could happen, for instance, if one or two problem classes in the test set have a lot of difficult instances – the single best choice (from training) might be an encoding whose size increases much faster than others and results in breaking the clause limit, memory limit, or time limit. In either of those three cases the penalty time of 10 times the timeout is then recorded. On the other hand, although the default choice (*Tree_Tree*) was not the single best on the training set, it performs well enough on the test set to avoid the heavy penalties.

An additional visualisation of selector performance is given in Figure 4.4, showing the number of instances solved as the time allowed is increased up to the timeout of 1 hour. With both splitting strategies we observe that the ML selectors follow a similar trajectory. It is interesting that in *split-by-instance* the default configuration is able to solve more of the easier problems, but is then overtaken by all the other selectors. In *split-by-class*, the single best is unable to catch up to the other selectors, for reasons discussed above. The default encoding is indeed a good choice for Savile Row as it keeps pace with "smarter" selectors until the 3000 seconds mark.

4.3.2 The Elusive Virtual Best for SAT Solving

Having evaluated the effectiveness of the approach, let us pause to consider the nature of the ideal we are pursuing, namely the *Virtual Best*. The intention is to train a machine learning system to predict the encoding configuration which would match the virtual best performance for that instance. It turns out that determining the virtual best is not straightforward. SAT solvers use randomness, for example to assign values to variables when restarting search. This stochastic element means that the same CNF formula may be solved in a different runtime (or number of steps) depending on the random seed. We can therefore can get different winning encoding configurations for one problem instance, as the CNF formula produced by one encoding may not always be solved more quickly that that produced by a different encoding.

We can observe the extent of the "fickleness" of the virtual best with two plots. In Figure 4.5 we see the variation in solving times for a random selection of problem instances and encodings. Each data point represents the 5 solving times recorded when solving a given SAT formula with Kissat. The plot uses error bars to show the longest and shortest solving times as well as the median time. We can see that in some cases, the difference in solving times can be more than an order of magnitude.

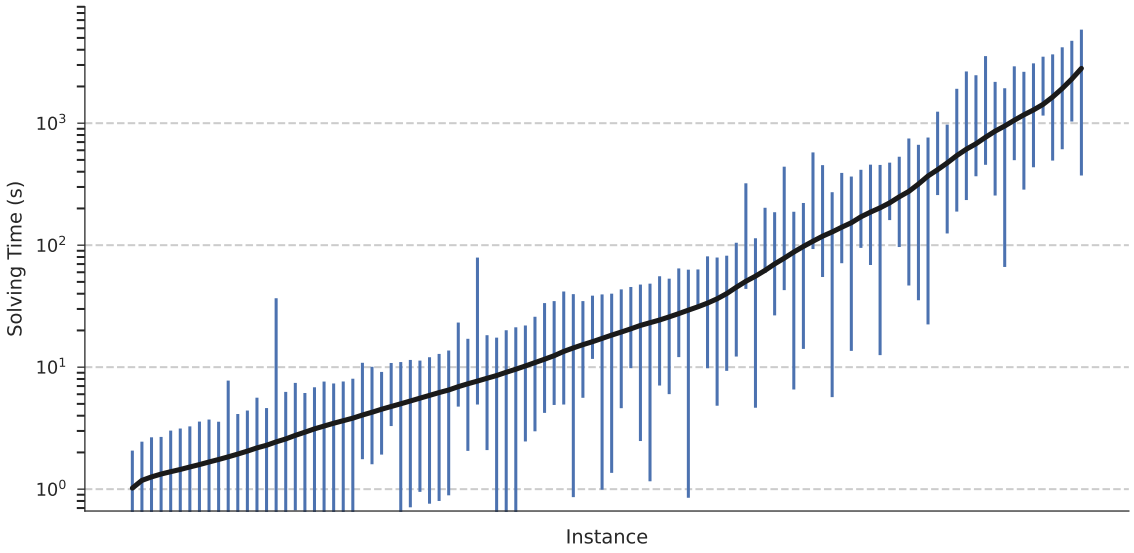


Figure 4.5: Variation in SAT solving time for a sample of 100 SAT formulae from the corpus. The vertical bars span the range of minimum to maximum for each formula; the solid black line shows the median time.

The second illustration of this difficulty is given in Figure 4.6, which shows the result of a small experiment. For a random selection of instances, I sample at random one of the solving runs for each configuration and record which configuration is the winner, i.e.

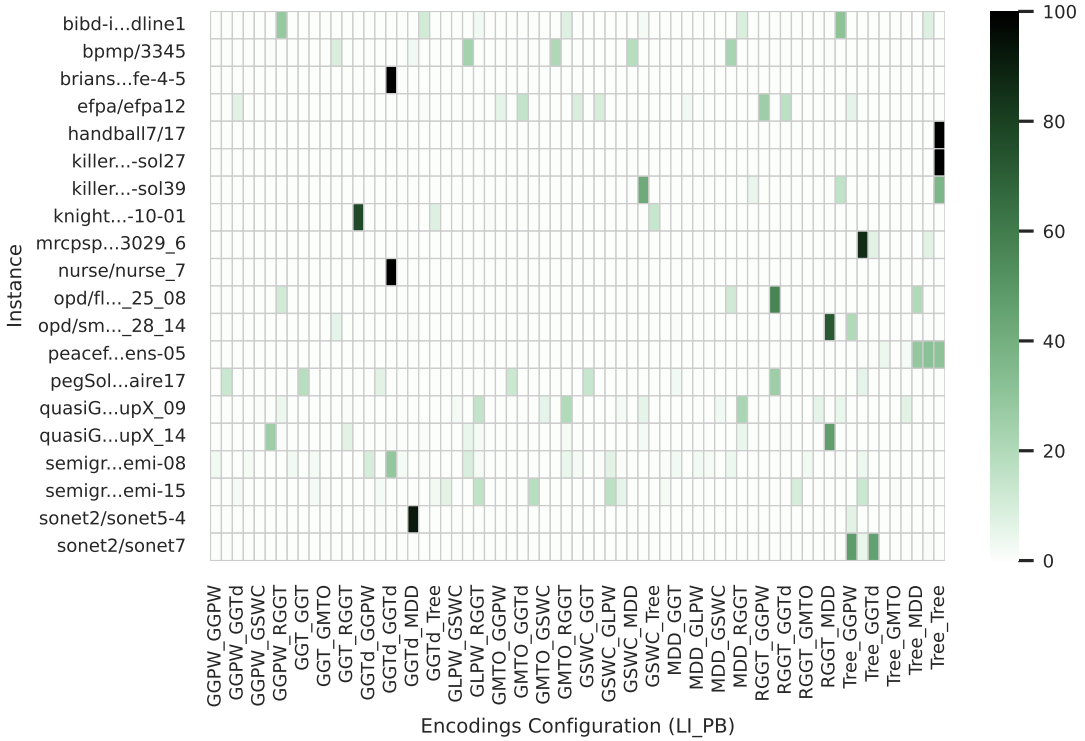


Figure 4.6: The variable identity of the virtual best encoding. For each randomly sampled instance, 100 "contests" are carried out, in which 1 runtime is sampled with replacement from the 5 available for each encoding configuration and the winning configuration recorded. The heatmap shows the distribution of such winners. Some encoding configuration labels are missing due to the limited space.

has the shortest solving time. The plot shows the distribution of the so-called virtual best encoding configuration after 100 such contests. Note that for some instances there is an almost even distribution among several encodings, whereas occasionally one encoding configuration consistently emerges as the best - for example in the fifth entry, the handball scheduling problem.

As explained earlier, in practice I narrow the choice by using a smaller portfolio of encoding configurations, and attempt to produce a "good" (rather than "the best") prediction by training classifiers on pairwise contests. In addition, I seek to mitigate the effect of the elusive virtual best by running the solver 5 times for each encoding configuration and using the median runtime as the basis of the labels in the dataset. Nevertheless I hope this small exploration emphasises the challenge of the task at hand.

4.3.3 Analysis of the Configuration Space

To conclude the account of the empirical investigation let us briefly analyse the configuration space in which we are making encoding choices. I have argued already that the

task of selecting suitable SAT encodings is not just a simple classification task. The observations below may shed some further light on important considerations when selecting encodings for a set of problems.

Table 4.3: Summary of the 20 best encoding configurations across the problem corpus by two different criteria. *Left*: the encodings which are best most frequently, showing the number of "wins" and the mean runtime for the instances on which it wins; the mean is rounded to the nearest second. *Right*: the encodings whose allocated instances in the virtual best selection have the highest total runtimes, and their contribution as a percentage of the total VB runtime. Encodings appearing in both top 20 lists are highlighted in **bold** type.

Most Frequent Winners			Biggest Contributions to VB	
Encodings (LI_PB)	Wins	Mean Time	Encodings (LI_PB)	% of VB
Tree_Tree	73	46	**GGPW_GGPW**	23.2
RGGT_Tree	31	127	**RGGT_Tree**	5.7
GGPW_GGPW	26	611	**GPW_Tree**	5.4
MDD_Tree	26	30	**GGPW_GGT**	5.3
GGPW_Tree	25	148	GGT_MDD	5.0
GGTd_Tree	24	28	**Tree_Tree**	5.0
GLPW_Tree	21	7	**Tree_MDD**	4.3
GGT_Tree	21	29	**GGPW_RGGT**	3.3
GSWC_Tree	19	17	GSWC_GGT	2.8
Tree_MDD	19	153	GGPW_GMTO	2.7
GGPW_MDD	18	16	GLPW_MDD	2.6
Tree_RGGT	15	21	GSWC_GSWC	2.6
GMTO_MDD	14	57	GLPW_GGTd	2.3
Tree_GGPW	13	77	GGT_RGGT	1.6
Tree_GGTd	11	76	GSWC_GGPW	1.6
GGPW_GGTd	11	76	MDD_GGTd	1.6
Tree_GSWC	11	43	GGT_GGPW	1.5
GGPW_RGGT	10	223	GGTd_GSWC	1.5
GGPW_GGT	10	362	**Tree_GGPW**	1.5
RGGT_GGPW	10	7	MDD_GMTO	1.4

In Table 4.3 I list the 20 best performing encoding configurations across the entire cleaned corpus[3] using two different criteria. Firstly, according to how often an encoding configuration is the best available; secondly, calculating the proportion of the total VB runtime allocated to a configuration. For instance, we see that *Tree_Tree* is the clear winner in the former league table, with more than twice as many wins as the next entry (73 vs. 31). However, the instances on which it wins have a mean runtime of 46 seconds. We can calculate its contribution to the VB as roughly $73 \times 46 \approx 3400s$, approximate because the mean is rounded. On the other hand, *RGGT_Tree* wins fewer times but the relevant instances are almost three times harder with a mean runtime of 127 seconds, making a contribution to the VB of approximately $31 \times 127 \approx 3900s$.

[3] Here each problem instance in the corpus is considered once, rather than sampled as part of a test set. Recall also that the cleaned corpus only contains instances for which at least one encoding configuration terminates before the timeout, so the PAR10 penalty does not apply here.

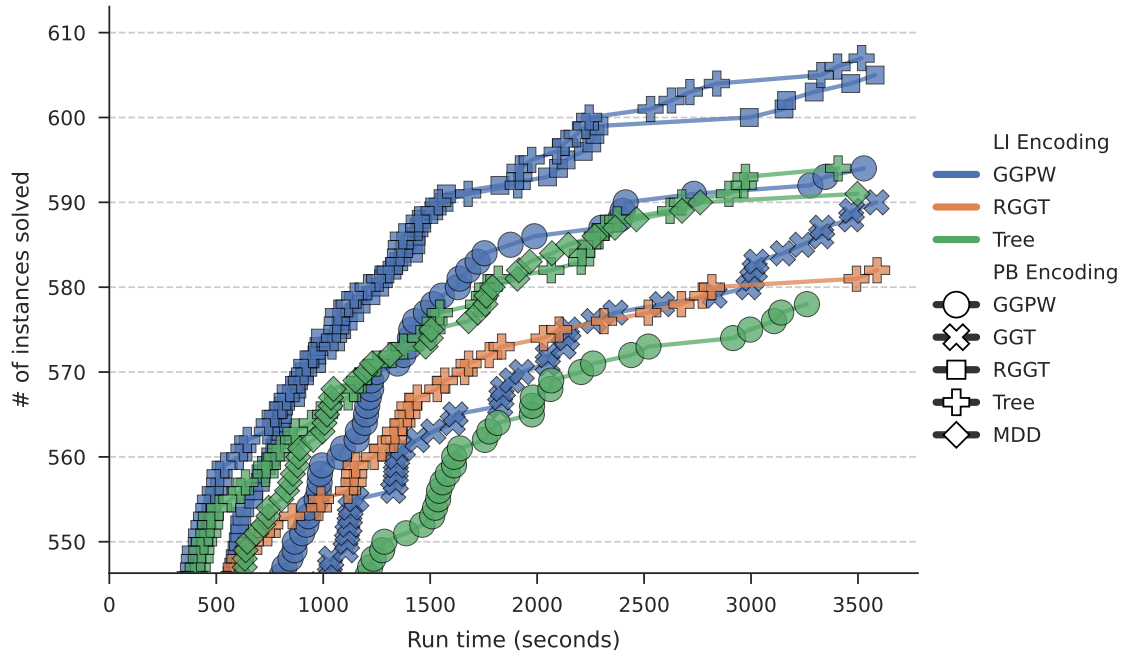


Figure 4.7: Performance profile of selected encodings on the entire corpus, showing how many problems can be individually solved within a given time, up to the timeout of 1 hour. The LI encoding is represented by the line colours and the PB encoding by the marker shape. We show the encoding configurations which appear in the top 20 both in terms of their contribution to VB and the number of times they are the best (see Table 4.3).

Figure 4.7 shows the performance profile of the encoding configurations which appear in both top 20 lists, and highlighted in bold in Table 4.3. As before, we see that GGPW is a great choice for LI constraints, appearing in 4 of the top 8 combined performers, whereas there is greater variety in the PB encodings in the best configurations. Figure 4.7 also demonstrates that these "top" encoding choices are excellent all-rounders – the top 2 are each able to solve over 600 of the 614 instances in the cleaned corpus (in which every instance is solvable within the timeout by at least one encoding configuration from the full set of 81).

In terms of prediction accuracy, choosing *Tree_Tree* most often makes sense, and indeed this is the default encoding provided by Savile Row. This is an excellent choice if the task is to solve very many problems, each of which are individually relatively easy, i.e. would solve in under a minute on our hardware. If, instead, the task is to solve "harder" problems, then, at least according to our corpus of problems, GGPW is a good choice for both LI and PB constraints. Recall that all encodings except *Tree* take advantage of AMO groups to reduce the size of the SAT encoding – another important consideration when selecting an encoding.

4.4 Summary

This chapter has described the design of LEASE-PI, a machine learning system with which I attempt to select good encoding configurations for constraint problems, where "good" encodings lead to shorter solving times. This is initially done on the basis of generic instance features. I go on to document an experiment which rigorously examines the effectiveness of the system and then to evaluate the results for many variations of the ML setup.

Addressing the research question posed, I find that the ML predictions do lead to performance improvements, even when predicting for unseen problem classes.

We also see that SAVILE ROW's default configuration is competitive, and we consider how difficult it is to even decide what the virtual best encoding configuration is due to the random elements in SAT solving. Finally we survey the collection of available encodings, discussing more broadly what makes a good encoding choice.

In the next chapter I will show that performance improvements are achievable by using features which specifically consider the relevant constraints. I will go on to discuss in more detail the complex issue of determining which features are important.

Chapter 5

Learning Using Specialised Constraint Features

♫ I wish I was special ♫

Radiohead

Research Question 3

Can the quality of encoding selection be improved by the use of features which are specific to the relevant constraints?

In the previous chapter I showed that it is possible to outperform the Single Best classifier by training ML models to predict an encoding choice. This worked very well for known problem classes and with some success even for unseen problem classes. However, there is an obvious limitation to the approach. The choice being made concerns which encoding to use for pseudo-Boolean (PB) or linear integer (LI) constraints but the features used do not directly measure anything about those actual constraints.

The next stage, outlined in this chapter, is to design and extract features which directly measure aspects of the constraints we are interested in.

The chapter contains:

- a description of a new set of features introduced to capture characteristics of the LI and PB constraints in a CSP instance
- results of repeating the experiments introduced in the previous chapter with the new features

- a comparison with AutoFolio, an established off-the-shelf algorithm configuration tool
- a discussion of feature importance

5.1 New Features to Describe Pseudo-Boolean and Linear Integer Constraints

The *f2f* and *f2fsr* features introduced in Section 4.1.2 can identify and describe CSPs well enough to make good predictions about what SAT encodings could work well for new instances and even new classes of problems. We now consider how to specifically describe the PB and LI constraints within a CSP.

5.1.1 The Anatomy of PB and LI Constraints

Recall that we describe an LI constraint as having the form

$$\sum q_i x_i \diamond k$$

where each x_i is a decision variable with a finite integer domain, q_i is an associated integer coefficient and $\diamond$ is a comparison operator. PB constraints are a special case of LI constraint where all the decision variables are Boolean. As explained in the previous chapter, I extract features just before Savile Row's internal model is encoded to a Boolean formula. By this stage Savile Row has transformed any PB or LI constraint so that the comparison is either $=$ or $\leq$.

For our purpose then a PB/LI constraint can be said to be characterised simply by the list of coefficients and the upper limit k. However, the possibility of partitioning the $q_i x_i$ terms into *at-most-one* groups brings an additional factor into consideration.

At-Most-One Groups

In the background chapter on SAT encodings we saw that when the decision variables in PB constraints are also subject to At-Most-One (AMO) constraints, it is possible to produce more compact encodings. These encodings can lead to much faster solving in some cases than encodings which are not AMO-aware. In Savile Row, LI constraints can be represented as PB constraints with AMO groups.

Table 5.1: 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 and the rows are numbered for easier reference. Features are extracted separately for the PB and LI constraints in an instance.

	Aspect of constraint	Aggregate
1	Number of (PB or LI) constraints (`count`)	*not applicable*
2	Number of terms (`n`)	min, max, mean, median, IQR, skew, ent, sum
3	Sum of coefficients (`wsum`)	sum, skew, IQR
4	Minimum coefficient (`q0`)	min, mean
5	Maximum coefficient (`q4`)	max, median, mean
6	Median coefficient (`q2`)	median, skew, ent
7	IQR of coefficients (`iqr`)	median, skew
8	Coefficients' quartile skew (`skew`)	mean, min, max, ent
9	Distinct coefficient values (`sep`)	mean, max
10	Ratio of distinct coefficient values to number of coefficients (`sepr`)	mean, max
11	Number of At-Most-One groups (AMOGs) (`amogs`)	mean
12	Mean size of AMO group (`asize_mn`)	mean
13	Mean AMOG size ÷ number of terms (`asize_r2n`)	mean
14	Mean maximum coefficient size in AMOGs (`amaxw_mn`)	mean
15	Skew of maximum coefficient in AMOGs (`amaxw_skew`)	mean, ent
16	Upper limit (k) (`k`)	mean, median, max, IQR, ent, skew
17	$k \times$ number of AMOGs (`k_amo_prod`)	mean, IQR, ent

It is important to take into account the AMO groups when describing PB / LI constraints, because they can have a big effect on the resulting encoding. For instance, the PB constraint $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 \leq 8$ has a left-hand side which could sum to 15 if all decision variables were true. However, if a partition with AMO groups $\{x_1, x_2, x_3\}, \{x_4, x_5\}$ existed then the maximum sum now reachable would be 8, requiring fewer SAT variables to cover all possible sums. Indeed AMO-aware encodings could even declare a PB trivially true in light of the AMO groups, for example if the sum of the maximum coefficients from each group is less than the upper limit for the whole PB.

Some of the features I introduce attempt to capture some of the interaction between the AMO groups, the coefficients, and the upper bound k.

5.1.2 The Features

The parameters of constraints considered for the PB-related features are summarised in Table 5.1 and fall into some broad categories.

Aspects 2-8 address the distribution of coefficients (also often referred to as *weights*) within a PB constraint by considering: how many terms exist (n), the sum of all the coefficients (wsum), the middle and extreme values (q0, q2, q4), the spread of coefficient values (iqr) and how skewed the coefficients are (skew).

Parameters 9-10 look at how many separate coefficient values exist as a number (sep) and as a proportion of the number of terms (sepr). These parameters distinguish between constraints where most coefficients are the same and those where the decision variables have a variety of associated coefficients.

In rows 11-15 I consider AMO groups: firstly how many exist (amogs) and how big they are (asize_mn, asize_r2n), and secondly how big the maximum weights in each group are (amaxw_mn, amaxw_skew). The maximum coefficients in each group determine the biggest overall total theoretically obtainable.

The parameter set also contains the upper bound k and a result (k_amo_prod) which estimates the maximum encoding size for some encodings under consideration.

These parameters are obtained for each PB/LI constraint in an instance – these are then aggregated for the whole instance using the functions listed in the right column of Table 5.1. This is done separately for the PB and LI constraints in an instance. Although ultimately they are represented in the same way, it may be important to record whether the constraint came specifically from a pseudo-Boolean constraint or from a more general LI constraint such as a sum. When the 17 parameters are aggregated across the instance using the various summary functions, the resulting number of features is 45 collected separately for the LI and PB constraints separately, leading to a total of 90 features.

5.2 Extended Results with Generic and Specialised Features

The 90 features described above form the *lipb* feature set. I have created a further featureset *combi* which is the union of the specialised *lipb* and generic *f2fsr* features. I now present the results of training with the new features alongside the previous results which used only generic features.

Here is a summary of all four featuresets used in the experiments presented in this chapter:

f2f The features produced by the fzn2feat tool on FlatZinc output from Savile Row.

f2fsr The same features extracted directly from Savile Row.

lipb The specialised set of features described in Section 5.1.

combi The union of *lipb* and *f2fsr*

5.2.1 Results

In Table 5.2 I present the updated performance results to include the predictions made using the new featuresets. The table reports the results in the same way as Table 4.2 in the previous chapter, giving the PAR10 times for the predicted encodings as a multiple of the virtual best (VB) time. Once again I present for reference the single best (SB) obtained by selecting the encoding configuration which yielded the lowest total time on the instances in each training set. The results using the default (Def) encoding of *Tree_Tree* are also shown.

On Performance

A striking initial observation is that in the *split-by-class* setting, the new featuresets become the best performing for all but one setup. Even when splitting by instance, the new featuresets improve the performance in seven out of 11 cases. Using the new features, we go from being able to close 46% of the VB-SB gap for unseen classes to 59%, placing us closer to the VB than to the SB and considerably outperforming the default setting.

The PAR10 performance is shown using boxplots in Figure 5.1, once again giving us the distribution of total times per test set, this time incorporating the new features too. We also observe the number of timeouts on the right-hand side. The new featuresets seem to have minimal impact in the *split-by-instance* setting, but are helpful when splitting by class, bringing down the mean time per test set and reducing the number of timeouts. Combining the specialised features with the generic ones seems to in fact reduce performance – the mean PAR10 time and the mean number of timeouts is lower when using *lipb* than using *combi*, although this is explained by the one outlier (with 20 timeouts!) which is produced by the *combi* predictions but avoided by *lipb*. Overall the evidence suggests that using the specialised features to predict encodings for unseen problem classes is a more robust approach than relying on generic instance features.

Recall that in the previous chapter I selected one setup as preferred; this is the fourth entry in the results table (Table 5.2) using pairwise RF classifiers with custom scoring and sample weighting. In the expanded results with the new featuresets we see that this setup is still the best performing choice on unseen problem classes and competitive on seen

Table 5.2: Total PAR10 times over the 50 test sets as a multiple of the virtual best configuration time. The best time for each combination of setup and splitting method is shown in **bold**. The predicted runtimes include feature extraction time. In the setup details, the selection is either done via *pairwise* voting or a *single* multi-label classifier; *DT* is a decision tree classifier, *RF* is random forest, *GB* is a gradient boosting classifier; *Co/Sep* shows whether LI and PB encodings were selected separately or as a combined choice; *SW* means sample weighting is used; *CL* indicates custom loss used in cross-validation; *Tuning* refers to the number of cycles of hyperparameter tuning.

Split-by-Instance Reference Times

Virtual Best	Single Best	Default	Virtual Worst
1.00	13.87	17.84	123.70

Split-by-Instance Predicted Times

Setup Details					Features			
Selection	Co/Sep	SW	CL	Tuning	f2f	f2fsr	lipb	combi
Pairwise RF	co	-	-	50×15	7.99	**5.63**	5.87	5.86
Pairwise RF	co	✓	-	50×15	6.33	5.36	5.10	**4.99**
Pairwise RF	co	-	✓	50×15	6.01	4.74	**4.57**	4.75
Pairwise RF	co	✓	✓	50×15	5.40	4.69	**4.43**	4.57
Single RF	co	✓	✓	750	3.91	**3.70**	3.77	3.81
Single RF	co	✓	✓	60	3.95	**3.70**	3.98	3.83
Single RF	co	-	-	750	10.34	9.64	9.11	**8.90**
Pairwise RF	sep	✓	✓	$50 \times 15 \times 2$	6.53	6.67	**5.60**	5.81
Pairwise DT	co	✓	✓	50×15	6.00	6.06	7.14	**5.99**
Pairwise GB	co	✓	✓	50×15	4.25	3.97	4.14	**3.58**
Single GB	co	✓	✓	750	4.25	**3.44**	4.12	3.69

Split-by-Class Reference Times

Virtual Best	Single Best	Default	Virtual Worst
1.00	25.40	17.15	160.96

Split-by-Class Predicted Times

Setup Details					Features			
Selection	Co/Sep	SW	CL	Tuning	f2f	f2fsr	lipb	combi
Pairwise RF	co	-	-	50×15	15.07	15.01	14.17	**12.55**
Pairwise RF	co	✓	-	50×15	16.80	14.90	13.21	**12.77**
Pairwise RF	co	-	✓	50×15	16.42	15.30	14.97	**11.16**
Pairwise RF	co	✓	✓	50×15	15.69	15.19	**11.00**	11.84
Single RF	co	✓	✓	750	20.59	19.15	**13.17**	14.53
Single RF	co	✓	✓	60	22.01	19.49	13.56	**13.52**
Single RF	co	-	-	750	19.72	19.93	**15.18**	16.50
Pairwise RF	sep	✓	✓	$50 \times 15 \times 2$	16.91	14.10	**12.02**	12.71
Pairwise DT	co	✓	✓	50×15	17.63	**14.05**	20.1	16.04
Pairwise GB	co	✓	✓	50×15	19.26	17.45	14.51	**12.25**
Single GB	co	✓	✓	750	24.54	19.76	**14.84**	15.02

classes. It's worth noting that the setup which predicts separately for PB and LI also does well on unseen problem classes. As mentioned in the previous chapter, this latter setup could become even more significant if we extended the number of constraint types we want to deal with – this setup would potentially scale better as each classifier would continue working in one dimension of the configuration space rather than three or more.

Let us return to the "survival" plot, this time with all four featuresets in Figure 5.2, showing the number of instances solved as time is increased. With both splitting strategies we observe that the featuresets emerge in the order *lipb*, *combi*, *f2fsr*, *f2f* with the *lipb*-predicted encodings enabling most instances to be solved within the timeout. When splitting by instance, the results for the new featuresets are very close to the original generic points; however, when splitting by class, there is a clear advantage for the specialised featuresets.

A further insight is provided by Figure 5.3 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, i.e. picking the best encoding available from the portfolio. In the split-by-instance scenario the prediction accuracy is fairly consistent across feature sets; however, for unseen classes we observe something unexpected. The *f2f* features lead to the most accurate predictions, but, as discussed above, the overall performance in terms of the resulting runtimes is considerably worse – this could be explained by the extra time required to output FlatZinc and then run the feature extractor. The specialised features enable the pairwise classifier to produce a more robust prediction, in the sense that even when the prediction made is not the absolute best encoding choice, the selected encoding tends to provide performance closer to the best.

On Encoding Choices

Figure 5.4 shows the frequency with which different encoding configurations are predicted. Recall that although we use a portfolio of 6 encodings, the portfolio is generated from the training set; consequently the portfolios are different across the 50 sets. The VB column shows a smooth distribution of ideal encoding choices from the full range of encodings available. In both splitting scenarios we observe five configurations preferred by the classifiers: *RGGT_Tree*, *GGPW_GGPW*, *Tree_MDD*, *GGPW_GGTd* and *GGT_MDD*. Additionally, in the split-by-class task *GGPW_Tree* is also used very often. The GGPW encoding for LI constraints is popular in these choices and can therefore be considered a very good single choice in many settings; remember that when illustrating the building of a portfolio using

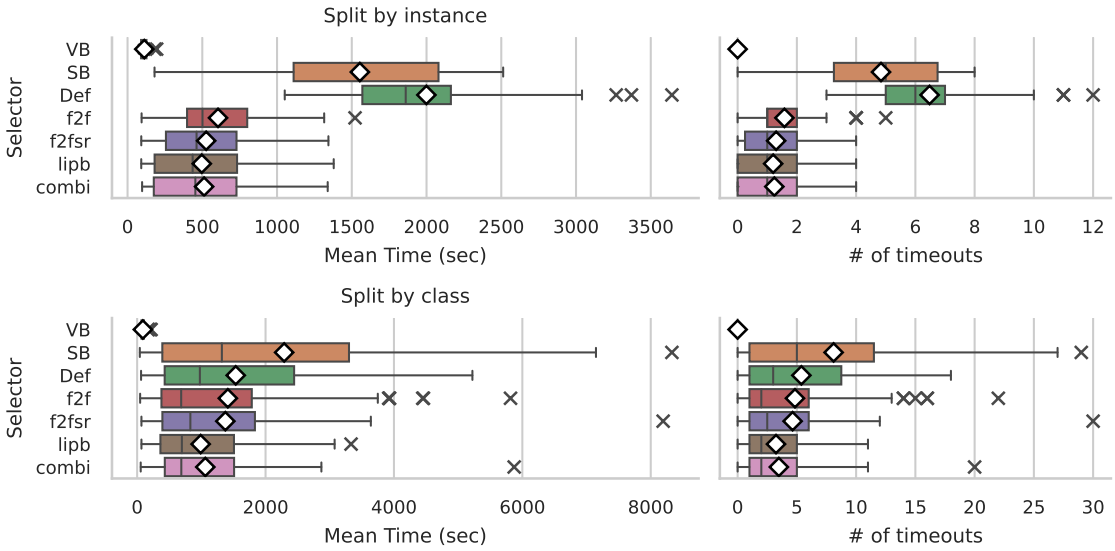


Figure 5.1: Prediction performance using all four sets of features against reference times. We show mean PAR10 runtime (left) and number of timeouts (right) per test set across the 50 cycles with our preferred setup (*pairwise random forests, sample weights, custom loss*). Outliers are indicated with crosses and represent values more that 1.5 × IQR outside the quartiles.

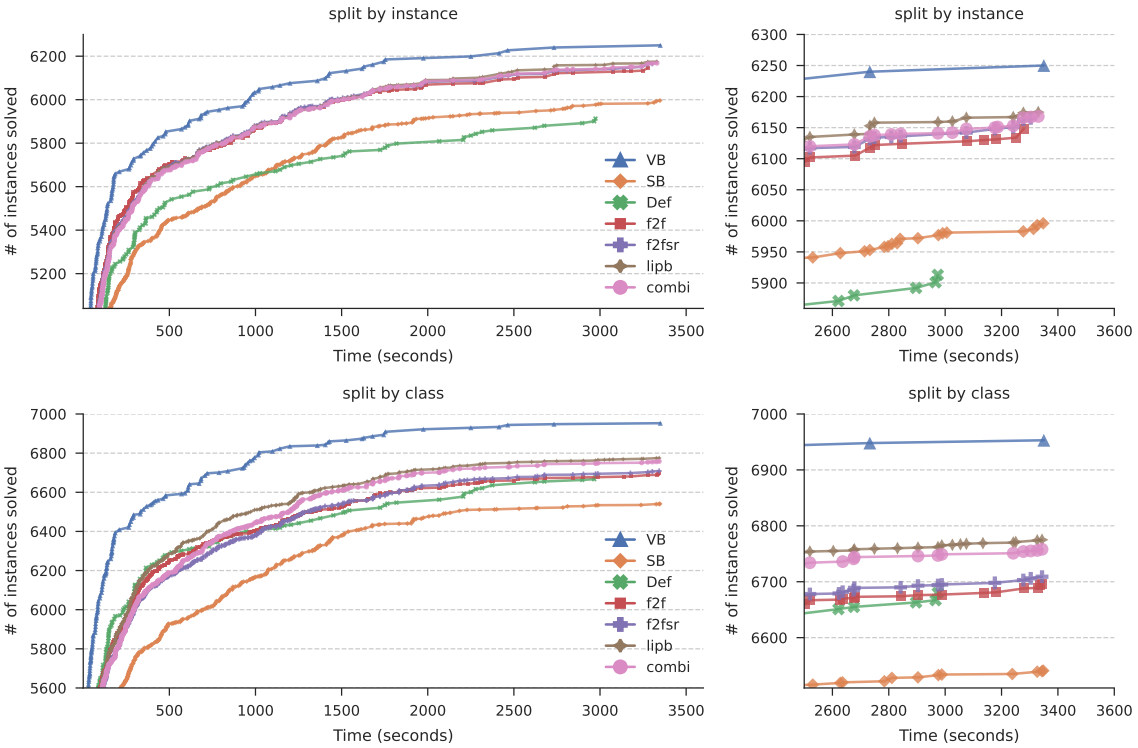


Figure 5.2: The number of problem instances individually solved within a given time for the reference selectors and our preferred predictor using different feature sets. The figures on the left show the full performance profile; on the right we zoom in to see how many instances are solved by the selectors as we approach our timeout limit of 1 hour.

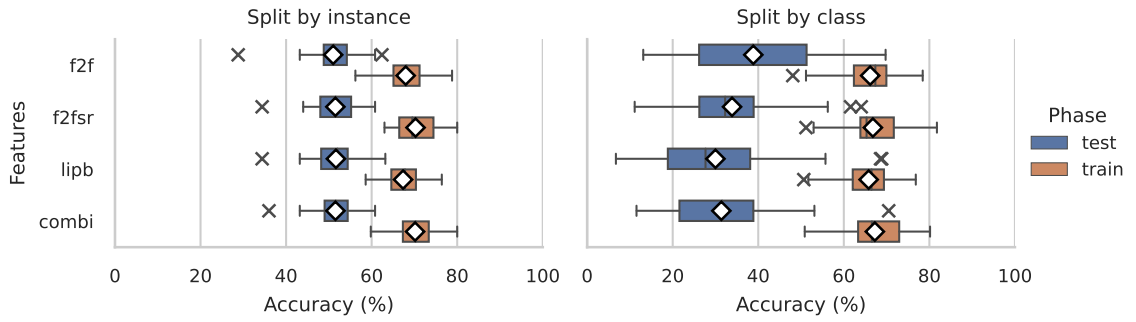


Figure 5.3: Distributions of prediction accuracy across the 50 *split, train, predict* cycles using our preferred setup.

the whole corpus in Figure 4.1, the single-choice winning configuration was *GGPW_Tree*. In the distribution of predictions made, there is more variety in the PB encoding selected, with five different choices featuring in the six top configurations mentioned.

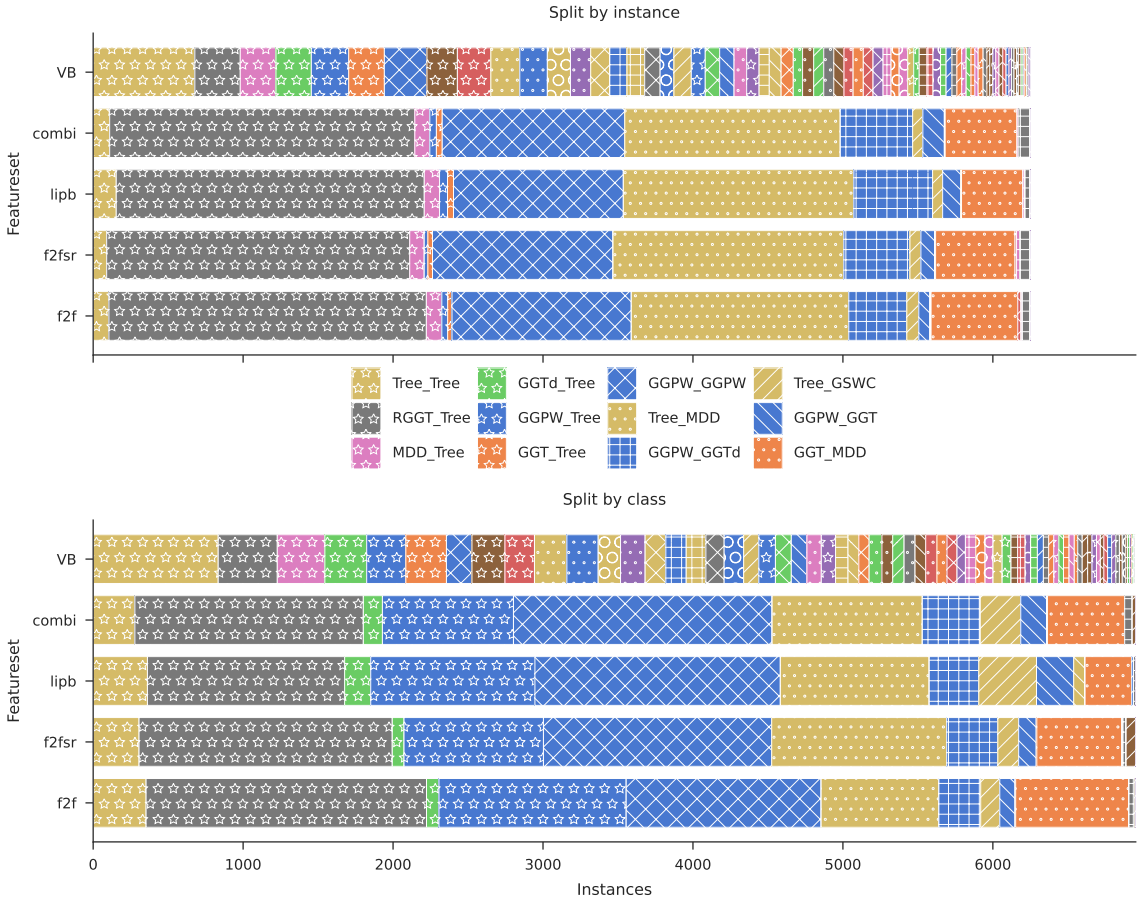


Figure 5.4: Frequency of each configuration (*li_pb*) selected across the 50 test sets when using each feature set with our preferred setup. We also show the virtual best (VB) configuration distribution for comparison. The colour indicates the LI encoding and the fill pattern shows the PB encoding. Only the top 12 most used configs (of 81 in total) are shown in the legend.

Statistical Significance and Effect

As well as comparing PAR10 times, it may be worth considering statistical tests to interpret the value of our approach.

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 50 mean times from the SB selector choices and our preferred selector using the *lipb* features. Using a two-tailed test, we obtain a p value of 6.5×10^{-5} (well below even a 1% significance level) and an effect size of -0.62 using the rank-biserial correlation method which would usually be interpreted as a medium to large effect.

The Vargha-Delaney A measure [90] provides another way to assess the effect size. When applied to our preferred setup and featureset, the Vargha-Delaney A value is very close to 0.5 (for predictions with *lipb* versus single best) if we apply it instance-by-instance to the nearly 7000 individual test runtimes – a figure which suggests no effect at all. If, instead, we take each of the 50 test runs in turn and consider the combined runtime, we obtain a value of 0.38, pointing to a small effect (the authors of the test indicate that a value below 0.36 would be a medium effect). However this VD-A analysis only considers whether a value drawn from one distribution is higher than from the other distribution on more occasions and as such has no sense of how much we might be "winning" or "losing" by. This argument is well made in [91] especially related to the sphere of search-based software engineering. This paper suggests that careful processing of the data is needed to avoid the trap of a completely wrong conclusion were one to use the VB-A measure naïvely.

5.3 Comparison with AutoFolio

To further contextualise the performance of LeaSE-PI, I compare with AutoFolio [83], a sophisticated algorithm selection approach which automatically configures algorithm selectors and "can be applied out-of-the-box to previously unseen algorithm selection scenarios." I use the latest version of AutoFolio available at the time of running the experiments (the 2020-03-12 commit which adds a CSV API to the 2.1.2 release) with its default settings. I use the algorithm selection component of AutoFolio to make a single prediction per instance, turning on the hyperparameter tuning option; I do not use its pre-solving schedule generation.

Table 5.3: Performance of LeaSE-PI against AutoFolio, given as PAR10 times over the 50 test sets as a multiple of the virtual best. The preferred LeaSE-PI setup is featured (using pairwise random forests, sample weighting and custom loss for tuning). The best time for each combination of setup and splitting method is shown in **bold**. The predicted runtimes include feature extraction time. *Tuning* refers to the number of cycles of hyperparameter tuning for LeaSE-PI and to the time budget given to AutoFolio.

Split-by-Instance Predicted Times

Setup Details					Features			
Selection	Co/Sep	SW	CL	Tuning	f2f	f2fsr	lipb	combi
Pairwise RF	co	✓	✓	50×15	5.40	4.69	**4.43**	4.57
AutoFolio	co	n/a	n/a	1 hour	22.19	24.18	**20.63**	21.02
AutoFolio	co	n/a	n/a	2 hours	26.71	27.59	22.63	**22.19**
AutoFolio	co	n/a	n/a	4 hours	22.76	25.40	**22.45**	23.47

Split-by-Class Predicted Times

Setup Details					Features			
Selection	Co/Sep	SW	CL	Tuning	f2f	f2fsr	lipb	combi
Pairwise RF	co	✓	✓	50×15	15.69	15.19	**11.00**	11.84
AutoFolio	co	n/a	n/a	1 hour	24.28	27.31	**24.21**	26.79
AutoFolio	co	n/a	n/a	2 hours	26.90	31.85	**25.26**	25.84
AutoFolio	co	n/a	n/a	4 hours	24.88	25.36	**23.66**	30.22

To compare as fairly as possible, I train AutoFolio on exactly the same training data, and test on the same test sets as above. The ML system I implement takes less than 5 minutes to train using 8 cores on the cluster, so I allow AutoFolio 1 hour on one core to tune and train. I also run it with a more generous budget of 2 hours and 4 hours to see if its performance improves. The runtime performance based on AutoFolio's predictions is shown in Table 5.3 alongside the preferred LeaSE-PI setups.

The LeaSE-PI predictions lead to better runtimes than AutoFolio's with five times faster solving on seen problem classes and more than twice as fast on unseen classes. AutoFolio is designed to be a 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 leads to marginal improvement with some feature sets, but in some cases actually results in worse performance, for example with *split-by-instance* and the *combi* features.

5.4 Feature Importance

I investigate the relative importance of instance features by using a measure called permutation feature importance. Breiman [60] 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. I implement this analysis but record the mean increase in PAR10 time when each feature is permuted, effectively giving the extra runtime cost when the feature is lost. Each feature is randomly permuted 5 times and the mean time increase recorded. The distribution of feature importance thus calculated is shown in Figure 5.5. I report on the *lipb* features and on the *combi* feature set which additionally contains the generic features from *f2fsr*. Only the top 20 features ordered by mean importance are shown.

Note that permutation feature importance is applied at prediction time to the test set, unlike the Gini (entropy) feature importance measure which is provided by the `scikit-learn` implementation of random forests but is calculated during training, as it observes how important the features are during the tree building phase. Other forms of feature importance exist too, for example one can compute a priori the correlation between each feature and the target value in the dataset if carrying out a regression.

We can see in Figure 5.5 that for both feature sets the median feature importance in the majority of cases is close to zero, but the mean importance varies considerably. 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 top features shown is able to prevent a costly wrong choice. The full extent of the variation of the mean feature importance is shown in Figure 5.6.

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_`), when it comes to predicting for known problem classes (i.e. split by instance). This is in keeping with the similar performance of the *f2fsr* and *libp* featuresets as shown previously in Table 4.2. It is likely that when splitting by instance the system is, to a large extent, recognising problem classes rather than picking out traits of PB/LI constraints.

When predicting for unseen problem classes, the proportion of PB/LI to generic fea-

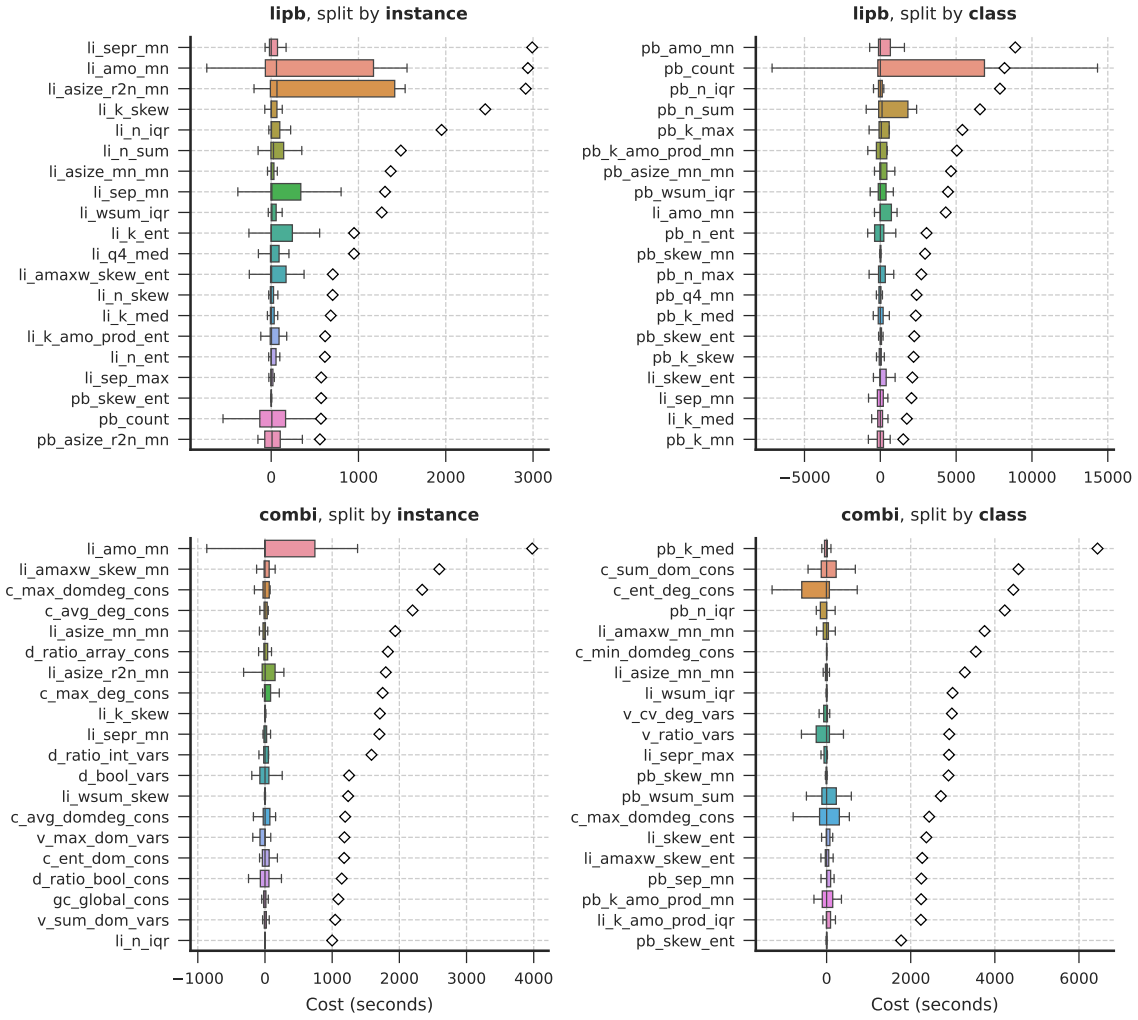


Figure 5.5: Permutation feature importance: increase in PAR10 time over 50 cycles. Top 20 features shown according to mean importance. Outliers are omitted, defined as being beyond 1.5 × IQR away from the box. The mean importance over the 50 cycles is shown by a diamond. Features beginning `li_` or `pb_` refer to LI/PB features as listed in Table 5.1; the other feature names refer to the generic instance features from the *combi* feature set.

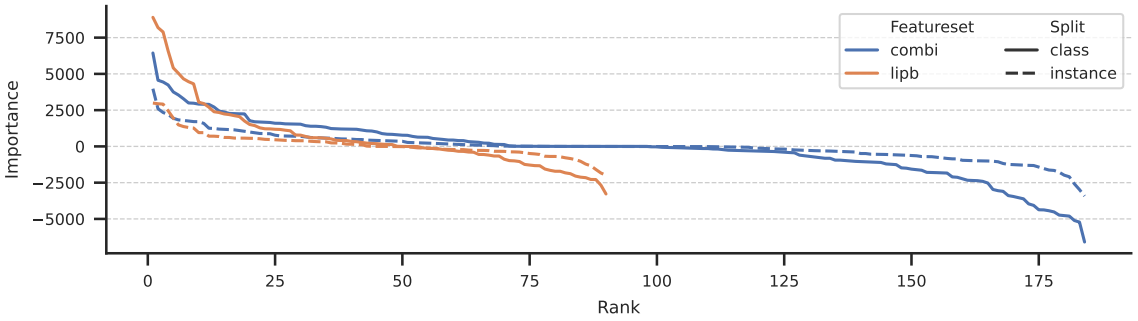


Figure 5.6: The mean permutation feature importance across the 50 cycles for every feature in the *lipb* and *combi* featuresets, from most to least important.

tures in the top 20 rises to 14:6, supporting the hypothesis that making choices about which encodings to choose for certain constraint types is better served by using features relating to those constraints in the problem instance.

Let us consider the importance of features related to PBs as distinct to LIs. Starting with the *split-by-instance* plot in Figure 5.5 (top left) we note that most of the top features relate to LIs (17 to 3); this is almost entirely reversed when splitting by class (16 to 4 in favour of PB features). One possible explanation is that there are more LIs on average than PBs in the corpus, so LI features could be more useful in recognising previousy seen classes; in the by-class case, the choice of PB is harder so PB-related features are more important here. We saw that the generic features performed competitively in the split-by-instance setting. However, when it comes to predicting for unseen problem classes, we know that the LI/PB related features are more discriminating as shown in the performance results (Table 4.2). We have also seen that there is more variation in the best PB encoding than the best LI encoding (with GGPW often winning for LI) and so it follows that the PB-related features are more prominent in this harder setting.

As a final discussion point relating to feature importance, we look in more detail at the top 20 features from the *lipb* featureset when used on unseen problem classes. We have already observed the disparity between the mean and median of the permutation feature importance. In Table 5.4 I list the top 20 features by mean and by median. A positive median value tells us that a feature is more often than not valuable in making good predictions. The mean indicates the overall contribution in a different way, i.e. how much time is lost on average per prediction batch across the 50 cycles when a feature is replaced with noise. The features highlighted in bold type appear in both top-20 lists and so can help to explain what kinds of features are most helpful. These mostly pertain to size, e.g. `pb_n_sum` is the total number of terms across all PBs, `pb_k_max` is the highest upper bound k, `pb_amogs_size_mn_mn` is the mean of the mean size of the AMO groups in PBs, `pb_count` is the number of PB constraints.

There are limitations to how much we can read into the permutation feature importance (PFI), in particular because of two factors: PFI considers features in isolation and we have quite a large number of features. A feature α may be discriminating but could be masked by another feature β with which it is highly correlated, so when we permute α's values, there may not be a great loss in prediction performance while the information from β remains. Thus we could wrongly conclude that α is not very valuable. We have also seen that the features in *libp* and *f2fsr* can give comparable prediction performance

Table 5.4: The 20 most important features in the *lipb* feature set by their mean and median permutation feature importance (PFI). Features which appear in both top-20 lists are highlighted in **bold**. These PFI values were obtained in the *split-by-class* task and are averaged over the 50 *split, train, predict* cycles.

Top 20 by Mean			Top 20 by Median		
	PFI (seconds)			PFI (seconds)	
Feature	Mean	Median	Feature	Mean	Median
pb_amogs_mn	8899.11	2.15	**pb_n_sum**	6573.40	114.78
pb_count	8190.50	3.33	**pb_k_max**	5412.10	68.36
pb_n_iqr	7889.19	0.69	**pb_amogs_size_mn_mn**	4661.48	30.20
pb_n_sum	6573.40	114.78	pb_k_amogs_prod_ent	1435.01	19.70
pb_k_max	5412.10	68.36	pb_wsum_sum	591.30	11.05
pb_k_amogs_prod_mn	5048.18	-0.36	**li_skew_ent**	2118.24	4.61
pb_amogs_size_mn_mn	4661.48	30.20	pb_n_min	323.83	3.66
pb_wsum_iqr	4462.16	-0.29	li_sepr_max	-1823.14	3.43
li_amogs_mn	4313.62	1.70	**pb_count**	8190.50	3.33
pb_n_ent	3048.54	3.01	**pb_n_ent**	3048.54	3.01
pb_skew_mn	2950.82	0.00	li_n_med	-164.57	2.22
pb_n_max	2703.56	-0.21	**pb_amogs_mn**	8899.11	2.15
pb_q4_mn	2390.52	-0.05	li_amogs_size_mn_mn	1168.76	2.11
pb_k_med	2337.28	-0.27	pb_k_amogs_prod_iqr	226.67	1.76
pb_skew_ent	2237.81	0.84	**li_amogs_mn**	4313.62	1.70
pb_k_skew	2192.99	-1.11	li_skew_mn	1213.57	1.13
li_skew_ent	2118.24	4.61	pb_k_ent	516.36	1.07
li_sep_mn	2055.11	0.05	li_count	422.29	0.87
li_k_med	1748.99	0.26	**pb_skew_ent**	2237.81	0.84
pb_k_mn	1513.33	0.04	**pb_n_iqr**	7889.19	0.69

(especially when splitting by instance) even though they consider different aspects of a CSP.

It is very difficult to draw strong conclusions about which features are the most significant. Many algorithms exist to aid feature selection before applying machine learning methods. Although further work in reducing the featuresets could be of value, I have shown that better predictions are achievable when using only the constraint-specific features in the split-by-class setting.

5.5 Summary

In this chapter I have extended LeaSE-PI by introducing features relating to PB and LI constraints. I have then presented the experimental results incorporating all four sets of features and shown that the specialised features improve performance considerably

when predicting for unseen problem classes, but make less difference when the constraint model has already been seen in training.

To assess whether LeaSE-PI is a competitive approach, I have compared with AutoFolio, an established and successful algorithm configuration solution. In the context of this particular task, i.e. selecting PB/LI SAT encodings, LeaSE-PI proved much better that AutoFolio. We also saw that the specialised features introduced in this chapter also led to better choices than generic features when used by AutoFolio.

I have calculated a measure of feature importance and analysed what it says about the relative importance of the LI-focussed features versus the PB-focussed ones in the context of my problem corpus. I have also discussed the difficulties present in trying to reliably identify the most important features.

The work presented so far still contains a major limitation: the choice of encoding is made once for all the LI constraints and once for all PB constraints in an instance. The next chapter explores how we might make a more fine-grained choice.

Chapter 6

IndiCon: Learning to Select Encodings for Individual Constraints

♫ You can go your own way ♫

Fleetwood Mac

Research Question 4

Is it practical to learn to set encodings for individual constraints within a problem instance? Does it lead to performance improvements compared to a single encoding choice per constraint type?

Constraint-specific features have increased our prediction power. However, so far I have chosen the same encoding for all constraints of one type in an instance. This could be problematic for at least two reasons:

1. 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.

2. So far I have aggregated features of individual constraints in order to produce a feature vector per instance. This means that if we had a few different profiles of constraint, say from different parts of the constraint model, we could be losing valuable information by aggregation.

Here I set out my attempt to select SAT encodings based on the features of individual constraints. I will refer to this system as LeaSE-IndiCon or, more briefly, just **IndiCon**.

This chapter comprises:

- an introduction to the motivation for IndiCon and an initial look at the expected challenges,
- an account of my attempts to overcome these challenges,
- a description of IndiCon i.e. my proposed method for learning to select encodings for individual constraints,
- the results of training and testing IndiCon with a variety of design choices, and
- an evaluation of the results

6.1 The Promise and Challenges for Per-Constraint Predictions

6.1.1 A Motivating Example

To illustrate how finer control of encoding choices might be useful in practice, let us consider a particular problem as an example. The multi-mode resource-constrained project scheduling problem (MRCPSP) has been studied thoroughly; it features in the PB(AMO) paper [43] we encountered in Chapter 2 and has extensive literature devoted to it, such as [92]. In MRCPSP a project to be planned has activities which can be carried out in different modes – the mode determines how long an activity takes and how much resource it consumes. The resources themselves can be renewable or non-renewable. The constraint model therefore contains two slightly different sets of pseudo-Boolean constraint, one for each class of resource. The fact that each activity can only be carried out in one mode leads to the presence of at-most-one (AMO) constraints which partition the pseudo-Boolean (PB) constraints. AMO constraints are also required to account for precendence relations, with tasks not being able to proceed concurrently.

A small extract of the constraint model written in Essence Prime is shown in Listing 6.1. The snippets show two important constraints which ensure that resource consumption does not exceed availability – this is done separately for the renewable and non-renewable resources. In this formulation, renewable constraints are replenished at each time step, so the demand on a renewable resource from all the active jobs at one time step must not exceed the resource available. Non-renewable resources are not replenished at any stage. The `sum` constraint in lines 4-9 is posted for every renewable resource and every time step to check that the resources demanded by the active jobs do

Listing 6.1: Selected extracts from the original constraint model written in Essence Prime for the multi-mode resource-constrained project scheduling problem (MRCPSP) – this model uses the sum constraint.

```
$ --- Renewable resources ---
forAll t : int(0..horizon) .
  forAll r : int(1..resRenew).
    sum(
      [
        active[j,m,t] * resUsage[j,m,r]
        | j: int(1..jobs), m: indexOf(durations[j,..])
      ]
    ) <= resLimits[r],

$ --- Non-renewable resources ---
forAll r : int(resRenew+1..resources) .
  sum(
    [
      resUsage[j,m,r] * (mode[j]=m)
      | j: int(1..jobs), m: indexOf(durations[j,..])
    ]
  ) <= resLimits[r],
```

Listing 6.2: Selected extracts from the adapted Essence Prime model for the MRCPSP, using the explicit amopb constraints

```
$ extra parameters to choose which PB(AMO) encoding to use
given enc_ren, enc_non : int(1..10)

$ --- Renewable resources ---
forAll t : int(0..horizon) .
  forAll r : int(1..resRenew).
    amopb(
      [
        [resUsage[j,..,r], [active[j,m,t] | m : indexOf(durations[j,..]) ] ]
        | j: int(1..jobs)
      ],
      resLimits[r],
      enc_ren
    ),

$ --- Non-renewable resources ---
forAll r : int(resRenew+1..resources) .
  amopb(
    [
      [resUsage[j,..,r], [mode[j] = m | m : int(1..maxmode)]]
      |j:int(1..jobs)
    ],
    resLimits[r],
    enc_non
  ),
```

not exceed the corresponding resource limit. Lines 12-18 introduce the `sum` constraints for each non-renewable resource. For both types of resource, the reader will notice that the resource consumption is dependent on `m`, i.e. the mode in which the job is done.

In Listing 6.2 the two constraints described above are replaced with explicit `amopb` constraints. Essence Prime's `amopb` constraint implements the PB(AMO) constraints and allows for a specific encoding choice to be made in the model. The three parameters to `amopb` are: a list of *(coefficient, Boolean decision variable)* pairs, the upper limit, and the SAT encoding choice.

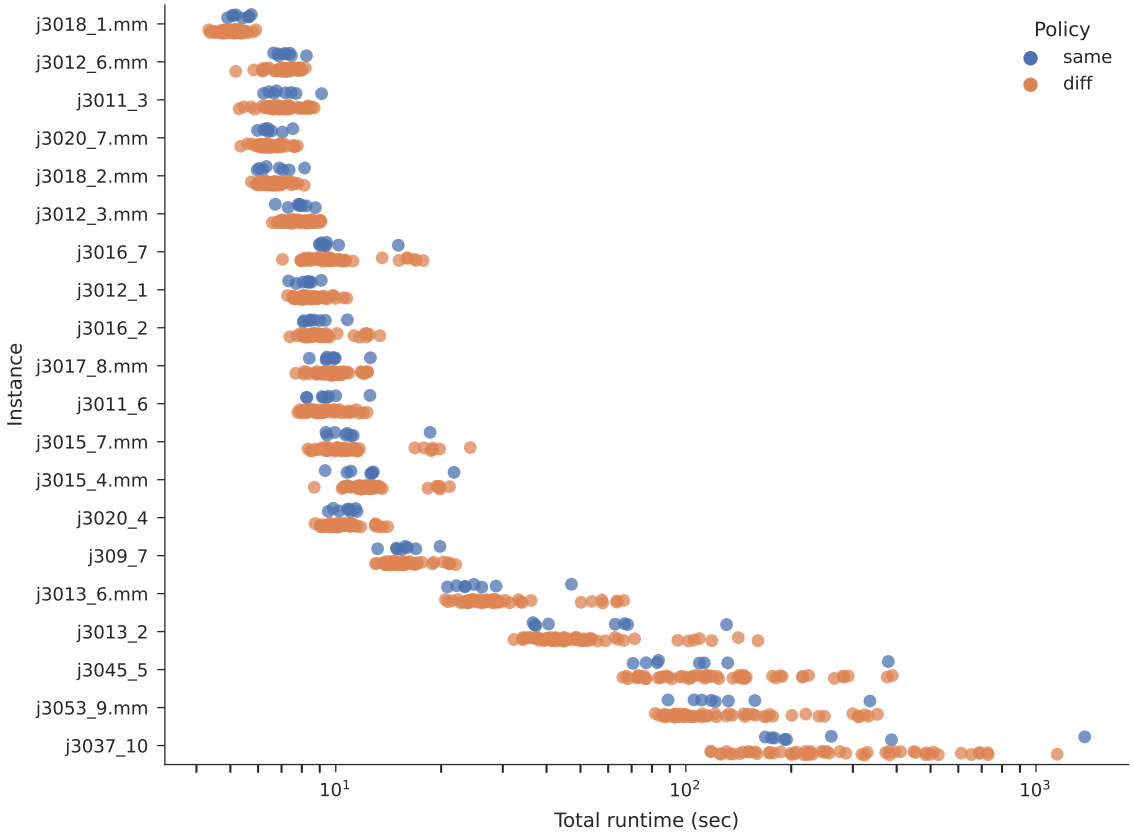


Figure 6.1: MRCPSP runtimes with same or different encodings for renewable and non-renewable constraints

I carried out an initial exploratory experiment, following the suggestion of my PhD advisor Peter Nightingale. I manually set the choice of PB(AMO) encoding in the constraint models for the renewable and non-renewable resources. I was able to use the 8 PB(AMO) encodings in Savile Row, giving 64 possible combinations of encoding. I took a random sample of 20 instances from the PSPLIB library of project scheduling problems [93] and ran each instance with all 64 configurations 5 times with the Kissat SAT solver, recording the median run time.

The results of this experiment are shown in Figure 6.1. The plot highlights the runtime of the encodings where both sets of PB constraints were assigned the same encoding in blue to distinguish them from the times when different encodings were set (in orange). If we consider the left-most blue and orange dots in each instance, we can see the best time possible when using a single encoding (like in LeaSE-PI) versus making separate choices. It appears that in some cases a sizeable improvement might be possible if separate encodings could be chosen. The gains from separate choices over a single choice are not huge in this particular experiment, but it suggests that there is room for improvement by allowing separate choices to be made. We must also consider several limitations of this experiment which could account for the fairly small potential gains shown:

- I only changed the PB constraint choice, but in other scenarios there might be other constraint types where we can adopt a more fine-grained choice.
- It was not easy to manually demand a *Tree* encoding for this experiment, so only the other 8 encodings available to LeaSE-PI were used. *Tree* is a very competitive encoding so could unlock further gains.
- The way I partitioned the constraints here was based on intuition and on the ease of manually parameterising the encoding used for the two different constraints as they're expressed in the Essence Prime model. Ultimately each individual constraint could be encoded using a different scheme, leading at least in theory to bigger potential "wins".

With the encouragement that partitioning the encoding choice can lead to performance improvements in a concrete case, we now turn our attention to some of the anticipated difficulties as I attempt to make decisions at a lower level, for each constraint rather than per instance. We will initially consider three aspects: obtaining useful training data, testing the system for the purpose of development and evaluation, and navigating some implementation issues for PBs in Savile Row.

6.1.2 Obtaining Timings to Build a Training Dataset

So far the thesis has looked at making one choice of SAT encoding per type of constraint per instance. Casting our minds back to Chapter 4 (Selecting Encodings using Generic Features), we saw that all the timing data was obtained up front. I ran all the instances in the corpus with every combination of encodings. This meant that when I was trying out a

new ML setup, the main time cost was how long it took to train the relevant ML models. Once they were trained, generating predictions is almost instantaneous for most models. From the predictions, I could then simply look up how long it took Savile Row + Kissat to solve any instance in the corpus using that predicted combination of PB and linear integer (LI) encoding.

It is however impractical to obtain timing data for every possible combination of encodings for each constraint in an instance. Many of the problems in my corpus have hundreds of PB or LI constraints in them – setting each constraint separately would give a huge number of combinations. For example, 9 encoding choices across 100 constraints would require us to time 9^{100} solver runs (if we ran with just one seed). Even if the mean runtime was 1 second, it would amount to 8×10^{87} CPU years! The original combined runtime of the timing data used in the previous two chapters is approximately 2 CPU years. We see then that in order to generate timing data on a practical scale, a more selective approach is required.

6.1.3 Evaluating Performance

As restated above, the evaluation step in the previous chapters was very quick because all possible combinations were already timed. When setting individual constraint encodings, we cannot practically pre-time all the combinations. The only way to test how well the ML selections work is to actually run the solving pipeline (Savile Row + Kissat) and record how long it takes. Purely in terms of computing resource, this is clearly a slower way to try out different ML setups.

In Figure 6.2 I attempt to illustrate this difficulty. We see that in the LeaSE-PI, only one large solving run is necessary in order to obtain all the timings. For every different ML setup that we want to investigate, some training time is required, but this tends to be much shorter, even with 50 trials[1]. The resulting predictions and evaluations are almost instant to compute. This all means that it is possible to quite quickly try out different learning setups, each new one perhaps requiring an average of around one hour of "wall time". However, when it comes to IndiCon, we need to solve the problems afresh with the individual encoding selections predicted by the ML, with several runs to mitigate stochastic variation in the SAT solver and once again with 50 different test/train splits to avoid evaluating an unrepresentative set of results.

[1] Recall that I did 50 trials to obtain a distribution of performance results over the random train/test splits

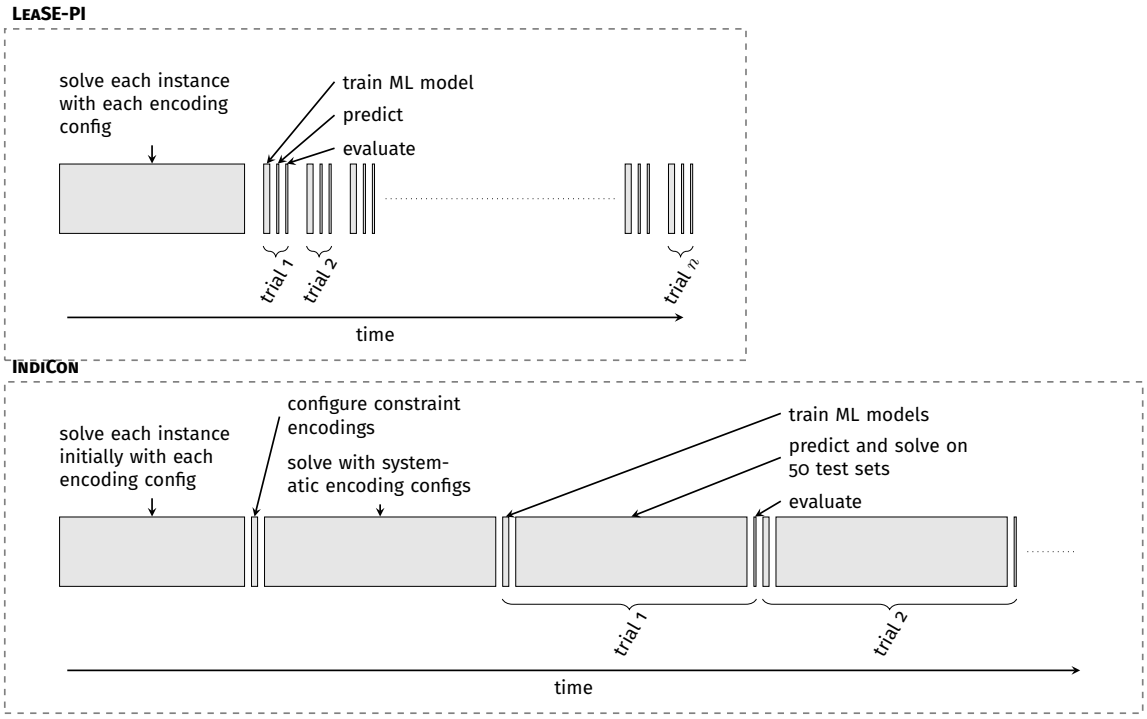


Figure 6.2: Timing challenges in evaluating performance of different setups. Top: LEASE-PI required one large solving effort to obtain all the timings, the ML training, prediction and evaluation phases were relatively quick for each setup. Bottom: to test each setup, SAVILE ROW must be run with the predictions to obtain the solving times for evaluation.

The time-consuming aspect of this investigation is compounded by two further factors, the details of which will be explained later in this section. Firstly, the PB and LI constraints are trained for separately, so the configuration of encoding choices per constraint and accompanying timing runs need to be done separately. Secondly, all the training relies on the initial per-constraint encoding choices - this configuration is very important, but every time it is changed all the subsequent training and testing would need to be done again.

6.1.4 To Tree or Not to Tree, or Not All PBs are Equal

As we saw in previous chapters, SAVILE ROW implements the TREE encoding for PB/LI constraints and this encoding is very competitive, coming out as the best choice most often both in the PB and LI context for the varied corpus used in Chapters 4 and 5. However, in SAVILE ROW this encoding is implemented in a slightly different way to the 8 PB(AMO) encodings mentioned in the previous two chapters and which were added into SAVILE ROW more recently.

The PB(AMO) constraints described in [43] are implemented assuming that normalisation steps have been taken. These steps are detailed in the paper and they ultimately

ensure that the constraint is in the form $\sum_{i=0}^{n} q_i x_i \leq K$, with positive coefficients q_i.

Savile Row can encode a sum of the form $\sum_{i=0}^{n} q_i x_i = K$ directly using the Tree encoding, but in order to use the PB(AMO) encodings, such a sum has to be broken into two separate less-than-or-equal constraints. This will have implications for how we extract features of PB/LI constraints and how we implement the ML-predicted choice at runtime.

In this section I have laid out some of the barriers in the way of building on LeaSE-PI to develop an ML-based system for predicting SAT encodings for individual (PB and LI) constraints. In the following sections I explain how I take on this challenge.

6.2 Method

Having discussed the various considerations and challenges faced, I now present my method for setting up IndiCon. Figure 6.3 gives an overview of the steps involved; if we compare to the steps required for LeaSE-PI (as shown back in Figure 4.2), we note two major differences, which have already been commented on in this chapter. Firstly, the labelled training data cannot be generated directly from the initial timing runs. Instead, we need to somehow produce per-constraint labels. Secondly, the results for analysis cannot be obtained by simply referring to the existing timings and applying the predictions.

The IndiCon process described in this section is applied either in the context of PB or LI constraints. Later on in this chapter I present an experiment which combines predictions for both constraint types, but it is important to understand that, in contrast to the approach used in LeaSE-PI, for this implementation of IndiCon, I try to learn to predict encodings for one constraint type. I have applied the entire process separately to PB and to LI constraints.

Let us begin with a high-level summary of the steps involved, as a companion to Figure 6.3, using the same numbers as the diagram for the steps and referring to the lettered labels for any data consumed or produced.

1. We start with a corpus of problems (A). We solve the instances initially with a single encoding choice per constraint type (as in previous chapters) and record the timings (B). These will help us identify good default encoding choice for the instance.

2. We extract features of each individual PB or LI constraint in the problem instances (C).

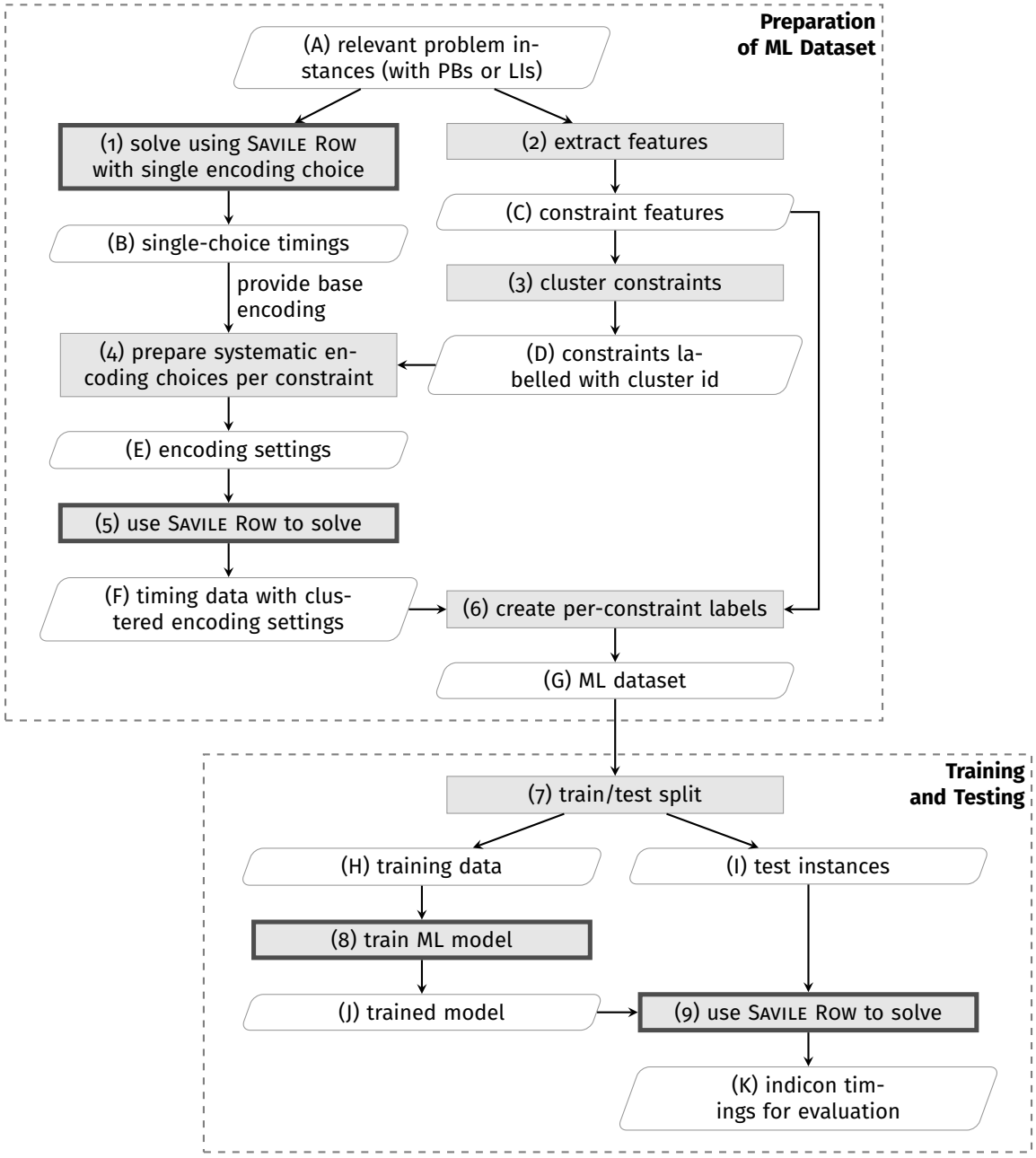


Figure 6.3: The steps involved in IndiCon. The slanted boxes with rounded corners represent data; the grey rectangles show processes. Processes which may take a long time have been drawn with a thicker border. Boxes have been labelled with numbers 1-9 for processes and letters A-K for data to make it easier to comment on the diagram in the text. The stages have been separated by dashed boundaries into the initial ML dataset preparation stage and the split/train/test phase which is repeated several times for different ML setups, and run with 50 different random seeds for each setup.

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. More details are given later in Section 6.2.4.

5. Each problem is solved using Savile Row 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 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 several 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 apart.

8. An ML model is trained (J) to predict per-constraint encodings.

9. To evaluate performance, we run the instances in our test set (I), consulting the ML model (J) from Savile Row to decide which encoding to use for each individual constraint, and recording the time taken (K) in order to analyse the performance resulting from the particular setup being used.

Through the rest of this section I present the details of the steps introduced above.

6.2.1 Corpus

I begin with the the same corpus of problems used in Chapters 4 and 5. I attempt to expand the corpus by considering all the problem classes in the PB(AMO) paper [43] on which Savile Row's implementation of PB encodings is based. MRCPSP already features in my corpus, but for the other problem classes it has proven difficult to obtain usable instances. This is either because there were no models available in the Essence Prime constraint modelling language, or because the instances available in that dataset are solved in such a quick time by Kissat that they are not challenging enough to make a contribution to this study.

Table 6.1: The corpus of problems used for IndiCon. The problems are mostly the same as in Table 4.1, but with extra instances for some problem classes. We show the name of the problem class, followed by the number of instances (n) and the mean number ($\bar{c}$) of PB or LI constraints per instance of each problem class, rounded to the nearest integer.

		$\bar{c}$				$\bar{c}$	
Problem	n	PB	LI	Problem	n	PB	LI
killerSudoku2	50	2473	194	magicSquare	6	232	57
nurse-sched	50	207	0	golomb	6	59	58
carSequencing	49	1024	0	peacefulQns3	6	0	6
knights	44	255	505	qGroup5Idemp	6	880	0
langford	39	231	0	qGroup6	6	756	0
opd	33	36	103	qGroup7	6	756	0
knapsack	24	1	1	bacp	4	0	53
sonet2	24	10	1	qGroup4Idemp	4	756	378
immigration	23	0	1	waterBucket	4	0	69
bibd-implied	22	651	0	qGroup4NonIdemp	4	1641	378
efpa	20	244	0	qGroup5NonIdemp	4	825	0
handball7	20	894	1809	qGroup3Idemp	4	825	378
mrcpsp-pb	20	100	62	qGroup3NonIdemp	4	1641	378
n-queens	20	1859	0	jp-encoding	2	0	2631
bibd	19	537	0	sol-battleship	2	111	48
molnars	17	0	6	discTomography	2	600	0
briansBrain	16	0	1	sokoban	1	0	36
life	16	0	786	plotting	1	1	28
n-queens2	16	361	0	sonet	1	4	1
bpmp	14	21	0	grocery	1	0	3
blackHole	11	303	0	sportsScheduling	1	249	96
pegSolAction	8	92	0	diet	1	0	7
pegSolState	8	92	0	contrived	1	0	6
pegSolTable	8	92	0	tickTackToe	1	9	21
peacefulQns1	7	0	5141	farm-puzzle1	1	0	3

One exception is the nurse scheduling problem. It is possible to access the full set of problems in NSPLIB [94] and convert them to Essence Prime format relatively easily. I obtain useful instances by converting the optimisation problem into a decision problem with a given upper bound. This allows us to run several hundred instances and obtain a random sample of 50 instances with a useful level of difficulty, i.e. which are solved in a non-trivial time but don't time out with the default SAT encoding settings in Savile Row.

The resulting corpus is shown in Table 6.1. Recall that IndiCon attempts to learn either about PB or LI constraints separately. This means that when setting up training for one of the constraint types, say PB, only problems which have some PB constraints (conversely LI) are used, so a smaller corpus of problems is available. For example, if we refer to

Table 6.1 we see that *opd* instances would feature when training for PBs and LIs, whereas *nurse-sched* will only be used to train for PBs and *bacp* only for LIs.

6.2.2 Building on *lipb* to Extract Features of Individual PB Constraints

In Chapter 5 I define some aspects of PB/LI constraints which are then aggregated to make the specialised featureset *lipb*. In order to describe individual constraints, I have adapted the feature extraction code in Savile Row to output the features of individual constraints, indexed by their path in the constraint model. In addition to the constraint aspects introduced previously and discussed in Chapter 5 I also introduce some extra features. The full list of features used by IndiCon is shown in Table 6.2. The motivation behind the new features is briefly described below:

- `is_equality` records whether the constraint uses the $=$ comparator rather than being an inequality; this has implications for how the constraint is encoded, as mentioned above in Section 6.1.4. I will explain further below how constraints are identified as relevant by Savile Row and how the equalities are treated differently to inequalities.
- `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`.
- `amog_maxw_mn2k` is the ratio of the average maximum coefficient in any AMO group 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.
- `amog_maxw_sum_k_prod` (the sum of maximum coefficients multiplied by the upper limit k) is a rough predictor of the maximum number of clauses required for some of the encodings.

Note that in the feature descriptions, as in literature on PB constraints, the words *coefficient* and *weight* are used interchangeably, hence the use of *w* in many feature identifiers, e.g. `amog_maxw_sum` which could also be described as the sum of the maximum weights.

Below is a brief discussion of some technical implications of extracting IndiCon features.

Table 6.2: Individual Constraint Features used by IndiCon, with their identifier and brief description. The *New* column shows new features added just for IndiCon.

New	Feature	Description
	n	Number of terms
	wsum	Sum of coefficients
	q0	Minimum coefficient
	q2	Median coefficient
	q4	Maximum coefficient
	iqr	IQR of coefficients
	skew	Coefficients' quartile skew
	sepw	Number of distinct coefficient values
	sepwr	Ratio of distinct coefficient values to number of coefficients
✓	is_equality	Is it an equality constraint?
	k	Upper limit K
	amogs	Number of At-Most-One groups (AMOGs)
	amog_size_mn	Mean size of AMOGs
	amog_size_mn_r2n	Mean AMOG size ÷ number of terms
✓	amog_maxw_med	Median size of the maximum coefficient across AMOGs
	amog_maxw_mn	Mean size of the maximum coefficient across AMOGs
✓	amog_maxw_mn2k	The ratio of amog_maxw_mn : k
✓	amog_maxw_sum	Sum of the maximum coefficients in each AMOG
	amogs_maxw_skew	Skew of the maximum coefficient in AMOGs
✓	amog_maxw_sum_k_prod	amog_maxw_sum $\times k$

Identifying Pseudo-Boolean Constraints

We noted in the motivating example at the start of this section that a PB constraint can be declared in different ways in Essence Prime. In the two different versions of the model for MRCPSP we used firstly an implicit definition, using the `sum` constraint over expressions containing boolean decision variables, and secondly an explicitly constructed `amopb` constraint, which at the time of writing was an experimental addition in a development branch of Savile Row. The first method is the standard way to define a PB constraint, and generally speaking this is desirable as we would want the modeller to focus on describing their problem. The "modelling assistant" (Savile Row) recognises that the constraint in question can be expressed as a PB, potentially with its terms partitioned into AMO groups.

For IndiCon, we are interested in how Savile Row identifies PB constraints in two phases of operation: feature extraction and the actual encoding stage. For feature extraction, I already implemented code which identifies all constraints and the variables in their scope in order to produce the *f2fsr* generic feature set used in Chapter 4. Savile Row provides a method to identify constraints which end up being encoded as PB, so they can be examined and their features recorded. Recall that we extract features as late as possi-

ble before the SAT encoding phase, so Savile Row has carried out all simplifications and reformulations, meaning the details of any PB constraint at this stage are exactly what will be encoded into SAT. For the eventual encoding phase, I have modified Savile Row to set the intended encoding choice for each relevant constraint according to the selections received from calling IndiCon.

Dealing with (In)Equalities

The PB(AMO) encodings used in Savile Row rely on the constraint being in a normal form where all coefficients are positive and the comparator operation is $\leq$; full details are given in [43]. PB or LI constraints with the equality ($=$) operator need to be split into two constraints, one with $\leq$ and one with $\geq$. The latter needs transforming into the aforementioned normal form. Potentially we might want to encode these two resulting constraints using different encoding schemes.

To make matters even more interesting, Savile Row's native implementation of PB constraints (called *Tree* and described in more detail in [3]) deals with equality constraints in a slightly different way. *Tree* also creates two inequalities but is able to reuse variables across the two constraints, making it potentially a more compact encoding. As we saw from our results with LeaSE-PI, *Tree* is a very competitive encoding.

This then is how I extract features to support both the PB(AMO) and the *Tree* encodings. IndiCon records the features of the original PB constraint (whether an inequality or an equality). If it is an equality, it then instructs Savile Row to construct the two PB(AMO) constraints (unattached to the "live" constraint model). The features of these two constraints are then also recorded so that the ML model can predict their encoding in case the original PB constraint is to be encoded using PB(AMO)s rather than *Tree*.

6.2.3 Clustering the Individual Constraints

Because we are seeking to predict encodings per constraint, we need to obtain training data in a different way than was done for LeaSE-PI. Previously we timed solving runs for our corpus of instances setting the SAT encoding choices per-instance; this enabled us to assign a *best encoding* label to each instance on which to train our ML classifiers. Now that we seek to make predictions for individual constraints within instances, we need to adapt how we arrive at a *best encoding* label for a feature row describing a single constraint. I describe below the attempted approaches.

Early Attempt #1: Learning from Host Instances

A relatively straighforward initial approach is to use the timings information from LeaSE-PI, where the encoding is set once for all constraints of a given type. I then construct a training set where I label the target encoding for each constraint to be the best encoding available for the host instance that the constraint appears in. 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. In a sense the whole motivation for IndiCon is to challenge that assumption; nevertheless, it may still be a useful, albeit imperfect, way to begin to make predictions per constraint.

Early Attempt #2: Clustering within Instances

If, however, we assume that constraints of the same type occurring within a problem instance have different features, then it makes sense to try and set different encodings for different constraints. Given that it is not feasible to try out every encoding combination, I partition constraints within instances by using an unsupervised clustering algorithm. This way, if an instance contains more than one constraint of the type we're insterested in (in our case PB or LI), we can try different encoding choices for all constraints in a cluster and record the resulting performance. Now we can pick out the best encoding per cluster and associate that as the target label for all the constraints in the cluster as we're constructing our dataset.

One decision that needs to be made with this approach is how many clusters to use. We have noted already that different problems have vastly different numbers of constraints (whether PB or LI). Clearly no clustering is required where only one relevant constraint exists, but for problems with many constraints, the number of clusters must be somehow chosen. This number needs to balance finer granularity with practical feasability. Two clusters might be too few to really capture the different types of constraints in the problem, whereas ten clusters would require running thousands of different encoding configurations just for that one instance. For this *cluster-within-instances* approach I use up to 5 clusters to achieve this balance.

This method has been helpful initially to set up the entire IndiCon pipeline with clustering, but does not feature in the full investigation as the number of clusters is so arbitrary.

Clustering Across the Corpus

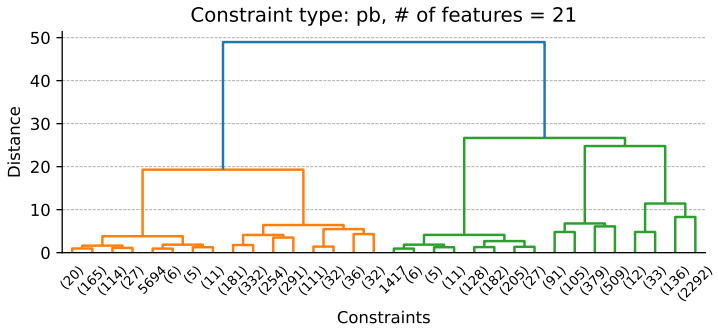
Clustering in the way described above is simple to implement, but treating each problem instance in isolation potentially suffers from some issues:

- we might miss out on patterns across the wider corpus
- we could be creating a false separation in some instances where in fact all the constraints might be near-identical, but simply with different decision variables in their scope
- we have to set the number of clusters once and apply that number to all the instances in the corpus. This arbitrary judgement may not reflect the distribution of constraints accurately.

Seeking to address the issues raised above, I have also implemented a clustering scheme which considers the corpus as a whole rather than partitioning constraints per instance. I use agglomerative clustering (introduced in Section 3.1.2) to determine the number of clusters globally by running the clustering algorithm and recording the distance values which cause clusters to be merged. The documentation for the agglomerative clustering implementation in `scikit-learn` recommends that the features are first scaled to the range $[0, 1]$. The distance between clusters is therefore defined as the Euclidean distance between the two feature vectors representing the location of the clusters.

We can observe the way the constraints in our corpus are clustered in Figure 6.4. Recall that each data point (in this case the feature vector of a constraint) begins by being its own cluster. As we increase the allowed distance between points in a cluster, we see clusters merging.

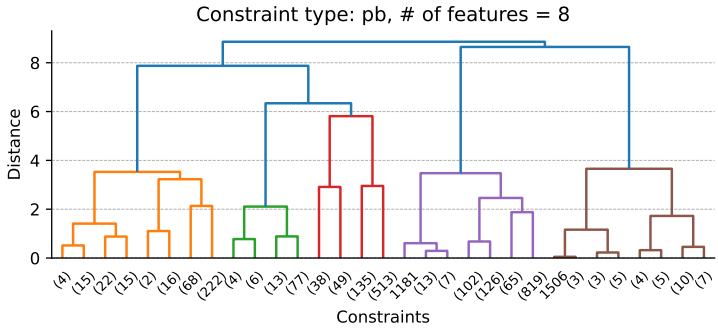
Let us look at the top dendrogram in Figure 6.4 as an example. We see that as the distance passes 12, 6 clusters become 5. The data remains split into 5 clusters all the way up until the distance is 19. After this point, the clusters keep joining quite quickly until we reach a distance of 27 and we're left with 2 clusters. These clusters remain distinct until we allow a distance of 49, at which point all constraints would fall into the same cluster. For our purpose of partitioning constraints in the problem corpus, we note that 2 clusters and 5 clusters remain stable for the largest ranges of distances (approximately 22 and 7 respectively). Although the data organises well into 2 clusters, I have chosen to use 5 clusters as this is also a relatively large gap between clustering points and it also allows us to try more configurations for our encodings while still being practical to investigate.



(a)

$\#c$	Δd
2	22.30
3	1.89
4	5.49
5	7.88
6	3.13
7	0.41
8	1.10
9	0.37
10	0.32

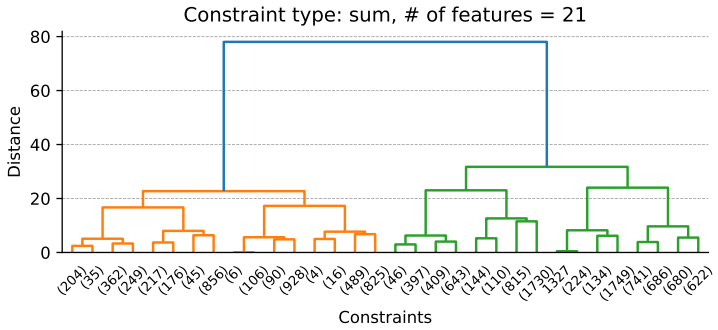
(b)



(c)

$\#c$	Δd
2	0.21
3	0.77
4	1.53
5	0.53
6	2.16
7	0.13
8	0.05
9	0.25
10	0.27

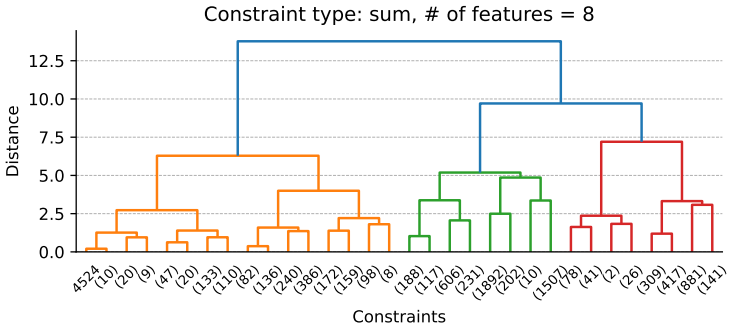
(d)



(e)

$\#c$	Δd
2	46.31
3	7.71
4	0.98
5	0.31
6	5.46
7	0.57
8	4.07
9	1.09
10	1.87

(f)



(g)

$\#c$	Δd
2	4.07
3	2.51
4	0.91
5	1.10
6	0.33
7	0.86
8	0.62
9	0.02
10	0.04

(h)

Figure 6.4: Dendrograms (on the left) showing the results of agglomerative clustering. The y-axis shows the Euclidean distance between clusters. Along the x-axis the bracketed labels indicate how many leaves would eventually be shown under that branch; labels without brackets give the index of individual constraints where the cluster size is 1. The right column shows the range of distance (Δd) spanned by each number of clusters ($\#c$), i.e. the heights of the vertical lines on the left-hand dendrograms. The clustering algorithm was run on the full set of IndiCon features (plots a,e) as well as on a reduced set of 8 most important features (c,g).

More Focussed Clustering with Fewer Features

The clusters of constraints used to label the training sets could potentially have a significant impact on the learning. One way to ensure that the clustering is as useful as possible is to focus on those features which are important for making good predictions.

In order to investigate this potential effect, I attempt to select a subset of features on which to base the clustering. I used the Sequential Feature Selection (SFS) method introduced in Section 3.1.2 and implemented by the `scikit-learn` library [6]. To keep options open as to which particular classifier to use later on, I employ SFS with four different classifiers which showed promise in preliminary experiments: Decision Tree, Random Forests, Gradient Boosted Trees and Multi-Layer Perceptron. I run SFS with a target of 10 features in "forwards" mode, i.e. starting with an empty featureset and adding features greedily to improve the classification accuracy.

Sequential feature selection works by evaluating the predictions made by a classifier – in order to do this we need a ground truth to be able to judge the classification quality. I use the same data as for LeaSE-PI, namely a single encoding choice per instances with aggregated features, i.e. the *lipb* featureset.

Once SFS identifies the 10 most important features for each classifier, I combine the results and keep the features which appear in more than one result. These are instance features, aggregated over all the constraints in the instance. One final necessary step then is to identify the aspect of a constraint which is represented in these "winning" features. For example `q2_skew` (the skew across all constraints of the median number of terms in each PB) is deemed important for classification of PB instances, so I select `q2` (the median number of terms in a PB constraint) as one of the IndiCon features to cluster on. The results of the SFS and the resulting individual aspects are shown in Table 6.3.

6.2.4 Generating Labelled Training Data

Once each individual constraint is associated to a cluster, we need a way to try out different combinations of encodings for those constraints in order to obtain runtimes – by comparing these runtimes we can then generate the target labels to train on. The exception is the first attempt described above, for which we can obtain our runtimes just as before (in LeaSE-PI), running each instance with each encoding across all constraints (5 times to mitigate any randomness in the SAT solver). However, we can view this approach as one particular case of the more general clustered approach where the number of clus-

Table 6.3: The results of sequential feature selection with four classifiers (decision tree, random forest, gradient boosted trees and multi-layer perceptron) for PB and LI constraints. The frequency column shows for how many classifiers each feature appear in the top 10 most important features. The root aspects of features which appear more than once are shown in **bold** type.

PB constraints		LI constraints	
Feature	Frequency	Feature	Frequency
q0_min	4	**q0**_min	4
q0_mn	4	**q0**_mn	4
q2_med	4	**amogs_size_mn_r2n**_mn	3
iqr_med	3	count	3
k_med	3	**iqr**_skew	3
q4_med	3	**k**_iqr	2
skew_min	3	**k**_skew	2
amogs_size_mn_mn	2	**q2**_skew	2
q2_skew	2	**sepr**_mn	2
q4_max	2	**skew**_ent	2
skew_max	2	amogs_maxw_skew_ent	1
amogs_maxw_skew_ent	1	amogs_maxw_skew_mn	1
k_amogs_prod_iqr	1	amogs_size_mn_mn	1
k_iqr	1	k_amogs_prod_ent	1
k_skew	1	k_ent	1
n_ent	1	n_ent	1
n_min	1	n_iqr	1
sep_max	1	n_skew	1
sepr_mn	1	sepr_max	1
		skew_min	1
		skew_mn	1
		wsum_iqr	1
		wsum_skew	1

ters happens to be 1. For the rest of this subsection we will focus on how to obtain timing data for clustered constraints.

Again we have to strike a balance between an exhaustive search and a practical experiment. Recall that we have 9 candidate encodings. If we had an instance with say 5 clusters of constraints, then the comprehensive approach would be to try every possible combination – however, that would amount to 9^5 combinations (plus we would want to solve it 5 times and take an average). A compromise is to choose a good default encoding for all constraints and then only change the encodings of one cluster of constraints in turn. This way the example above with 5 clusters would require $(1 + 8 \times 5) = 41$ combinations; this is made up from one combination where all 5 clusters are set to the default and then each of the 5 clusters is set to the 8 remaining encoding choices in turn. Working with 41 combinations is still a lot, but is feasible on a modern high-performance computing

Listing 6.3: A sample of how constraints are allocated

```
instance             conpath       cluster
-------------------  ------------  -------
mrcpsp-pb/j3028_9    T_0_1769            1
mrcpsp-pb/j3028_9    T_0_1770            1
mrcpsp-pb/j3028_9    T_0_1771            0
mrcpsp-pb/j3028_9    T_0_1772            0
            -- ROWS OMITTED --
n_queens2/queens-75  T_0_0__0_tmp        2
n_queens2/queens-75  T_0_0__1_tmp        0
n_queens2/queens-75  T_0_0               4
n_queens2/queens-75  T_0_100             2
```

cluster.

To illustrate this stage of IndiCon further, consider Listing 6.3 which shows a small sample of the information we have after clustering globally, i.e. across all instances in the corpus – each constraint in an instance is allocated to a cluster. Recall that where the PB or LI constraint is an equality we generate the two inequalities that would be needed if we use an encoding other than *Tree* – these extra constraints are identified with the suffix `_tmp`. From this information we can determine how many clusters are represented in each instance. We are then in a position to set the encoding for each constraint systematically, by trying each encoding one cluster at a time, while keeping the encodings of the other clusters set to a *base* encoding, which was the previously determined best encoding for that instance. We end up with several encoding configurations for each instance, a small sample of which is shown in Listing 6.4. We can see that for the `j3028_9` instance of MRCPSP, two of the global clusters feature and that the encoding deemed best overall for the PBs in that instance was GGPW - this becomes the *base* encoding. We end up with 17 different encoding configurations to solve and time. In the next example, we see the first few and last few entries for an instance of the N queens problem. In this case three clusters feature so we produce 25 configurations, i.e. the base configuration where all clusters of constraints are set to *Tree*, and the 8 other encodings in each of the 3 clusters.

At the end of this stage, once we have solved the instances in our corpus with all the systematic variations of encodings, we obtain a set of timings which allow us to generate training data for an ML model; more specifically, we can now assign a best encoding label to each individual constraint. For each constraint we filter the results to just consider the times when we were varying the encoding in the cluster that our constraint belongs to. From the fastest configuration of these, we select the encoding in our constraint's cluster "slot". So for example, looking again at Listing 6.4, if our constraint was in the `queens-75`

Listing 6.4: A sample of the individual constraint setting by cluster to produce timing information on which to train.

```
instance              label           base
--------------------  --------------  ----
mrcpsp-pb/j3028_9     ggt.ggpw        ggpw
mrcpsp-pb/j3028_9     ggth.ggpw       ggpw
mrcpsp-pb/j3028_9     ggpw.ggpw       ggpw
mrcpsp-pb/j3028_9     glpw.ggpw       ggpw
mrcpsp-pb/j3028_9     gmto.ggpw       ggpw
mrcpsp-pb/j3028_9     gswc.ggpw       ggpw
mrcpsp-pb/j3028_9     mdd.ggpw        ggpw
mrcpsp-pb/j3028_9     rggt.ggpw       ggpw
mrcpsp-pb/j3028_9     tree.ggpw       ggpw
mrcpsp-pb/j3028_9     ggpw.ggt        ggpw
mrcpsp-pb/j3028_9     ggpw.ggth       ggpw
mrcpsp-pb/j3028_9     ggpw.glpw       ggpw
mrcpsp-pb/j3028_9     ggpw.gmto       ggpw
mrcpsp-pb/j3028_9     ggpw.gswc       ggpw
mrcpsp-pb/j3028_9     ggpw.mdd        ggpw
mrcpsp-pb/j3028_9     ggpw.rggt       ggpw
mrcpsp-pb/j3028_9     ggpw.tree       ggpw

n_queens2/queens-75   ggt.tree.tree   tree
n_queens2/queens-75   ggth.tree.tree  tree
n_queens2/queens-75   ggpw.tree.tree  tree
n_queens2/queens-75   glpw.tree.tree  tree
n_queens2/queens-75   gmto.tree.tree  tree
n_queens2/queens-75   gswc.tree.tree  tree
n_queens2/queens-75   mdd.tree.tree   tree
n_queens2/queens-75   rggt.tree.tree  tree
n_queens2/queens-75   tree.tree.tree  tree
n_queens2/queens-75   tree.ggt.tree   tree
n_queens2/queens-75   tree.ggth.tree  tree
           --- ROWS OMITTED ---
n_queens2/queens-75   tree.tree.mdd   tree
n_queens2/queens-75   tree.tree.rggt  tree
```

instance and assigned to the first cluster, we would find the fastest mean runtime for the first 9 configurations as shown and select the name of the encoding in the first part of the label. By associating this best encoding label with the individual constraint features, we generate the training set.

Weighting

Having generated encoding configurations systematically, we still face at least two challenges when using timing results from solved instances to generate “best encoding” labels for each constraint.

Firstly, as we discussed in Section 4.1.4 when describing sample weighting, for some instances the choice of encoding may be more significant, so we may want to pay more

attention to those instances as their misclassification could be costlier. We can use sample weighting when training our classifiers to address this issue. We weight instances by their virtual best (VB) time to promote the "harder" instances. We can also weight those instances where the choice of encoding makes the most difference to runtime. For IndiCon I adapted the weighting formula to be

$$W = \log_2(2 + \frac{\mathrm{med}(T)}{\min(T)} \times \min(T)) = \log_2(2 + \mathrm{med}(T))$$

where T is the set of runtimes for the instance from all the different encodings, $\min$ is the minimum time (VB), and med is the median time. This differs from the sample weighting used in LeaSE-PI in a few aspects:

- Instead of using the ratio $\frac{\max(T)}{\min(T)}$ to signify how much an instance is affected by encoding choice, I have opted for $\frac{\mathrm{med}(T)}{\min(T)}$. The intuition behind this is to focus on the better half of the encodings, given that really bad encodings are very unlikely to be chosen given the pairwise arrangement of classifiers.
- I have switched the logarithm base from 10 to 2 in order to make the weighting less steep, hoping to learn from the medium-difficult cases a bit more than just from the very hard cases, which was likelier with the earlier base.
- The weight is now a float rather being restricted to integers. For LeaSE-PI, I thought discrete weightings would be useful for categorising and reasoning about the different tiers of importance, but I didn't anticipate needing to do this in IndiCon.

The resulting distribution of weights is illustrated in Figure 6.5 to give a sense of the range of difficulty and sensitivity in the problem corpus. Instances further to the right are harder, i.e. they have a longer virtual best runtime. Instances further up are more sensitive to the choice of encoding, i.e. they have more variation in runtime depending on the encoding chosen.

A second challenge is that there are vastly different numbers of constraints of a given type across the different problem instances so it's possible to end up with a very imbalanced training set. Suppose problem A has 5 PB constraints but problem B has 500. It seems undesirable to consider 100 times more training rows from problem B than problem A.

To go some way towards addressing this imbalance, I trim the dataset by dropping any duplicate feature rows from instances. In many instances, there are multiple constraints

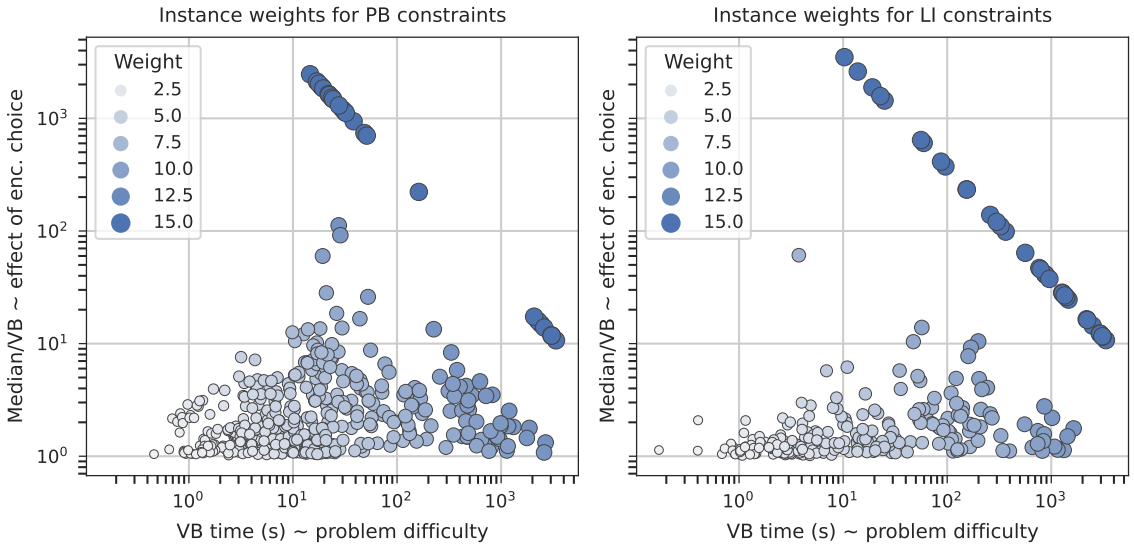


Figure 6.5: Instance weighting prior to training. The horizontal axis shows the VB time as a measure problem difficulty. The vertical axis represents difference in time between the median-performing encoding and the VB. The size and colour of marker indicate the resulting weighting for each instance. The left and right plots show subsets of problems which contain PB and LI constraints respectively.

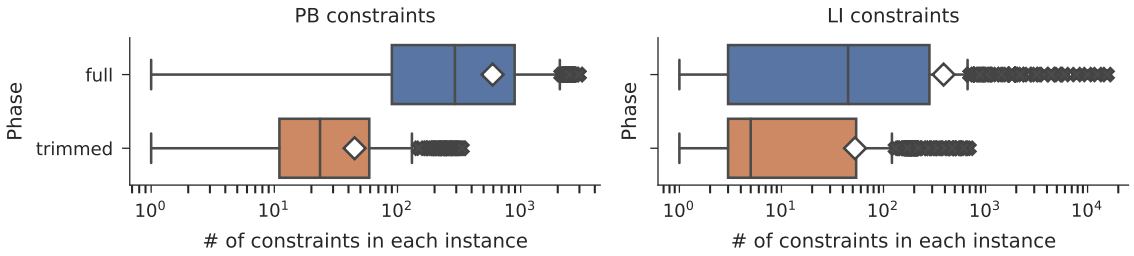


Figure 6.6: Distribution of number of constraints per instance before and after trimming to reduce duplicates. The x scale is logarithmic and the means are shown using white diamonds.

which are essentially identical apart from exactly which variables they cover. For training an ML model, these identical constraints can be collapsed into one row. The effect of doing this is shown in Figure 6.6. We see that the mean number of constraints per instance is reduced from 592 to 45 for PB constraints and from 388 to 53 for LI constraints.

6.2.5 Training Classifiers

For IndiCon we are training each ML model to choose between encodings for one type of constraint at a time, in contrast with the general approach for LeaSE-PI, where we consider combinations of encoding choices across two constraint types.

I choose the initial setup for IndiCon on the basis of what worked well in the previous chapter. We observed that on unseen problem classes the best performing setup over-

all was when random forest classifiers were trained on the specialised *lipb* featureset to choose between two encoding configurations at a time, which I called "pairwise" classification (Table 5.2). So pairwise classification by random forests is the starting setup, incorporating the weighting decisions outlined above (i.e. paying more attention to instances if they were harder or more sensitive to different encodings, and removing duplicated feature vectors within instances).

I have also tried other classification algorithms, again inspired by what seemed to work well for LeaSE-PI, namely gradient boosted trees, and basic decision trees. Additionally I group these three classifiers into a voting ensemble.

LeaSE-PI benefitted from a custom scoring function during hyperparameter tuning. I implemented the same function for IndiCon, the function simply reporting what the extra cost in seconds is of a particular set of predictions compared to the best known encoding configuration – for IndiCon this evaluation was done with reference to the cluster that each constraint belongs to. Initial experiments show that this custom tuning score function does not improve overall performance. I therefore try an additional tweak, which is to use the logarithm of the runtime penalty incurred. The idea here is to avoid being dominated by a small number of bad choices.

6.2.6 Testing IndiCon

As mentioned before and illustrated in Figure 6.3, we can no longer simply look up the timings that IndiCon's predictions would result in. Instead, the per-constraint predictions need to be implemented and the resulting encoding solved. Initially I test IndiCon on PBs and LIs separately given that the whole training workflow has been based on one constraint type. For each constraint type (PB or LI) the train/test split is carried out with 50 seeds. I train the classifiers on the training set and then run Savile Row with IndiCon on the test instances. I only use IndiCon to set the encoding type in question (e.g. PB constraints), leaving the other types of constraints to be encoded according to Savile Row's defaults.

I also go on to carry out two further comparisons in order to evaluate IndiCon in the context of the results of the previous system (LeaSE-PI).

Further Comparison #1: With LeaSE-PI

We saw in the previous two chapters that it was possible to achieve good speedups over the default and single best encodings with LeaSE-PI which selects one encoding for all the

Algorithm 6.1: The process of sampling suitable problems and trained ML models to test using IndiCon for both PBs and LIs in one solving run.

Data: A set S of $\langle I, s, t\rangle$ triples where I is a problem instance, $s \in \{1..50\}$ is the ID of the train/test split, and t is *True* if I is in the test set for split s; the per-instance feature set F for our corpus of problems; the number n of tests required

Result: A set C of $\langle I_i, M_i^{PB}, M_i^{LI}\rangle$ triples where I is a problem instance, and M^T is a model trained for constraint type T

Function `SampleSuitableTests`(S, F, n)**:**
- $J \leftarrow$ all instances in our corpus which have both PBs and LIs (according to F)
- $A \leftarrow \emptyset$
- **foreach** $j_i \in J$ **do**
  - $S_i^{PB} \leftarrow$ the split IDs in the PB trials where j_i was in the test set
  - $S_i^{LI} \leftarrow$ the split IDs in the LI trials where j_i was in the test set
  - $A \leftarrow A \cup \{\langle j_i, M(x), M(y)\rangle \mid x \in S_i^{PB}, y \in S_i^{LI}\}$ where $M(x)$ and $M(y)$ are the trained ML models from splits x and y for PBs and LIs respectively
- $C \leftarrow$ a uniform random sample of size n from A (with replacement)
- **return** C

constraints of one type in an instance. I have compared the performance of IndiCon with the best LeaSE-PI setup, namely pairwise random forests with custom loss and sample weighting using the *lipb* featureset (see Table 5.2).

The train/test splits are not identical for LeaSE-PI and IndiCon, because for IndiCon I prepare separate corpora for PB and LI constraints, pruning instances from each corpus which don't have the target constraint type. To enable the comparison, I sample with replacement ten PAR10 results from LeaSE-PI and ten from IndiCon for each instance in the IndiCon corpus, doing this separately for PB and LI. The mean PAR10 time across the sampled results is then used for each instance to compare performance.

Further Comparison #2: Both Types of Constraint

I have developed the IndiCon code in Savile Row to be able to set the encodings of both PB and LI constraints in one run by making separate calls to the relevant trained ML models. To make this experiment fair I ensure that I only consult trained models which have not seen the classes of constraint problems I am testing on. I test on problem instances which contained both PBs and LIs. I sample 3000 triples from the 11142 available combinations of problem instance, PB-trained model and LI-trained model. The process of preparing suitable test candidates for this combined experiment is summarised in Algorithm 6.1. Again, each of these Savile Row + IndiCon executions is run 5 times with different seeds.

A Note on Timings and Fairness

To make IndiCon work "live" and to be able to test it by actually solving instances and timing the process, we need Savile Row, which runs in Java to be able to interrogate the trained classifier for predictions. The training library is `scikit-learn`, a Python library. In its prototype form, for the purpose of the thesis, IndiCon calls Python from Java once for each type of constraint considered. This invocation takes a very long time, as python has to be started up and the relevant libraries and trained models loaded. This process can take several seconds even on the high-performance cluster.

This overhead could be addressed in many ways in order to make the whole pipeline more efficient. Here are some suggestions:

Client-server A server process could be running in the background, which has the relevant classifiers and libraries loaded. The server listens for requests from Savile Row containing a feature matrix and returns the recommended encodings.

Re-implement ML in Java The ML component could be written in Java and be directly part of the Savile Row code.

However, for the purposes of this investigation, I have addressed this issue by discounting the overhead in the following way:

1. Savile Row times how long it takes from starting the external process which calls Python to receiving the encoding back; let's call this time T_{total}
2. The python script `lease-indicon` itself records the time elapsed from once the libraries and ML model is loaded to when it has finished outputting the selections as text; let's call this time $T_{predict}$.
3. Savile Row calculates $T_{total} - T_{predict}$ and reports it as `LeaseTime` in its `.info` metadata file.
4. In the analysis, I deduct `LeaseTime` from the total time for a "run".

6.3 Results and Analysis

So far in this chapter I have set out the motivation behind selecting different encodings for individual constraints, described some of the inherent challenges and proposed a workflow for implementing and testing such a system. I now present the results for IndiCon

having tried out some of the different design choices described in the preceding Method section. These choices are summarised in Table 6.4.

Table 6.4: Summary of design choices in IndiCon at various stages

Name	Options	Description
Clustering scope	corpus, instance	Whether clusters are determined globally across the whole corpus or within each instance
Clustered on	all,sfs	Whether the clustering was done on all features or on a subset chosen by sequential feature selection
# of clusters	$1\ldots$	How many clusters to partition constraints into
# of features for clustering	$1\ldots$	How many features to consider when doing the clustering
Classifier	dtree, rf, gb, ...	Which classifier to use
Instance features	T/F	Whether whole-instance aggregated features are also added to the per-constraint feature vector
Classification type	single, pairwise	Whether to train a single multi-class classifier or use a pairwise voting approach
Custom loss	no, raw, log	Custom loss function for hyperparameter tuning

Once again, all experiments were carried out on the Viking high-performance cluster at the University of York with the same software versions and almost identical settings as for LeaSE-PI (see Section 4.2.1). Here is a brief re-cap:

- Viking has over 100 nodes, each with 2 Intel Xeon 6138 20-core 2.0 GHz processors; the memory limit per job is set to 6GB.
- Savile Row is run with AMO detection switched on, a SAT clause limit of 10 million, a timeout of 1 hour.
- Kissat `sc2021-sweep` is used again, 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.

6.3.1 Targetting PB and LI in Isolation

To evaluate the performance of IndiCon I compare to the virtual best time achievable by using a single encoding. We have discussed already that it is practically impossible to calculate a true virtual best because running every single combination of encodings for

every constraint in an instance is prohibitive in all but the simplest cases. Our reference is called *VB** in this chapter to emphasise that this is a single-choice virtual best. I have also calculated the single best (SB) result again, which is the result of choosing the one encoding which performed best on the training set.

The results for the setups I tried are shown in Table 6.5; as discussed this is just a small exploration of all the possible setup combinations that could be implemented. However, we can make some interesting observations even from these trials.

Constrasting the performance on instances with PB constraints versus the ones with LI constraints, we notice that IndiCon easily beats the single best (SB) choice with many of the setups, whereas in the LI case the best performing IndiCon setup (6.44×VB) is not as good as the SB encoding choice (4.53) from the training set applied to all LIs in the test instances. To some extent this echoes the findings in previous chapters that for LI constraints there is less meaningful complementarity in the choice of encodings, whereas for instances with PBs, the best encoding choice varies much more. The stark finding that IndiCon is so far unable to beat SB may unveil what could have been masked in the previous chapters by the fact we were predicting the encodings for both PB and LIs. Even when we used a separate ML classifier to choose the PB and LI encodings in LeaSE-PI we applied the choice together.

What is very surprising is that the two best-performing setups for PB do not make use of any clustering at all – they use 1 cluster (rows 1,2). In effect, we just label each constraint in our training set with the encoding that was best for the instance as a whole. Even when we consider the LI results, the setup which uses no clustering (row 1) is very close in performance to the top performer (row 5) with PAR10 × VB values of 6.70 and 6.44 respectively.

Once again random forests seem to give the best results, for both constraint types. It is encouraging to see that a simple decision tree can in fact perform very close to random forest of trees, as seen in row 1 in the PB table. In this setup (as in all the *pairwise* rows), the classifier learns to predict between two encodings. Decision trees are the most interpretable of the classifiers used and therefore it may be possible to extract meaningful rules for choosing between two encodings.

It is striking that the attempted enhancements do not seem to improve performance. We observe, for example that when the custom loss function is applied for hyperparameter tuning it degrades performance – see for example PB rows 4,5, and 6.

We also note that there is once again evidence in support of the pairwise classifica-

Table 6.5: INDICON performance for a variety of implemented 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 as a multiple of VB*. The tables show reference and performance times separately for the corpus with PB constraints (top) and with LI constraints (bottom). The best performance is indicated in **bold**. †This setup used 300 hyperparameter tuning cycles, whereas the rest of the setups used only 50.

Reference Times (PB)

Virtual Best*	Single Best
1.00	11.58

INDICON performance (PB)

	Setup Details						PAR10
Row	Clust. on	Clusters	Classifier	Inst. feats	Classif. type	Cust. loss	× VB*
1	all	1	DT	-	pairwise	-	5.69
2	all	1	RF	-	pairwise	-	**5.57**
3	all	1	RF	-	single	-	15.66
4	all	5	DT	-	pairwise	-	8.29
5	all	5	DT	-	pairwise	log	14.96
6	all	5	DT	-	pairwise	raw	8.49
7	all	5	DT	✓	pairwise	log	10.80
8	all	5	DT	✓	single	-	12.13
9	all	5	GB	-	pairwise	-	8.10
10	all	5	GB	-	pairwise	raw	10.65
11	all	5	RF	-	pairwise	-	8.46
12	all	5	RF	-	pairwise	raw	9.48
13	all	5	ensemble†	-	pairwise	-	8.58
14	all	5	ensemble	-	pairwise	-	10.05

Reference Times (LI)

Virtual Best*	Single Best
1.00	4.53

INDICON performance (LI)

	Setup Details						PAR10
Row	Clust. on	Clusters	Classifier	Inst. feats	Classif. type	Cust. loss	× VB*
1	all	1	RF	-	pairwise	-	6.70
2	all	6	DT	-	pairwise	-	16.24
3	all	6	GB	-	pairwise	-	11.12
4	all	6	GB	-	single	log	11.24
5	all	6	RF	-	pairwise	-	**6.44**
6	all	6	ensemble	-	pairwise	-	12.69
7	sfs	3	GB	-	single	log	13.49

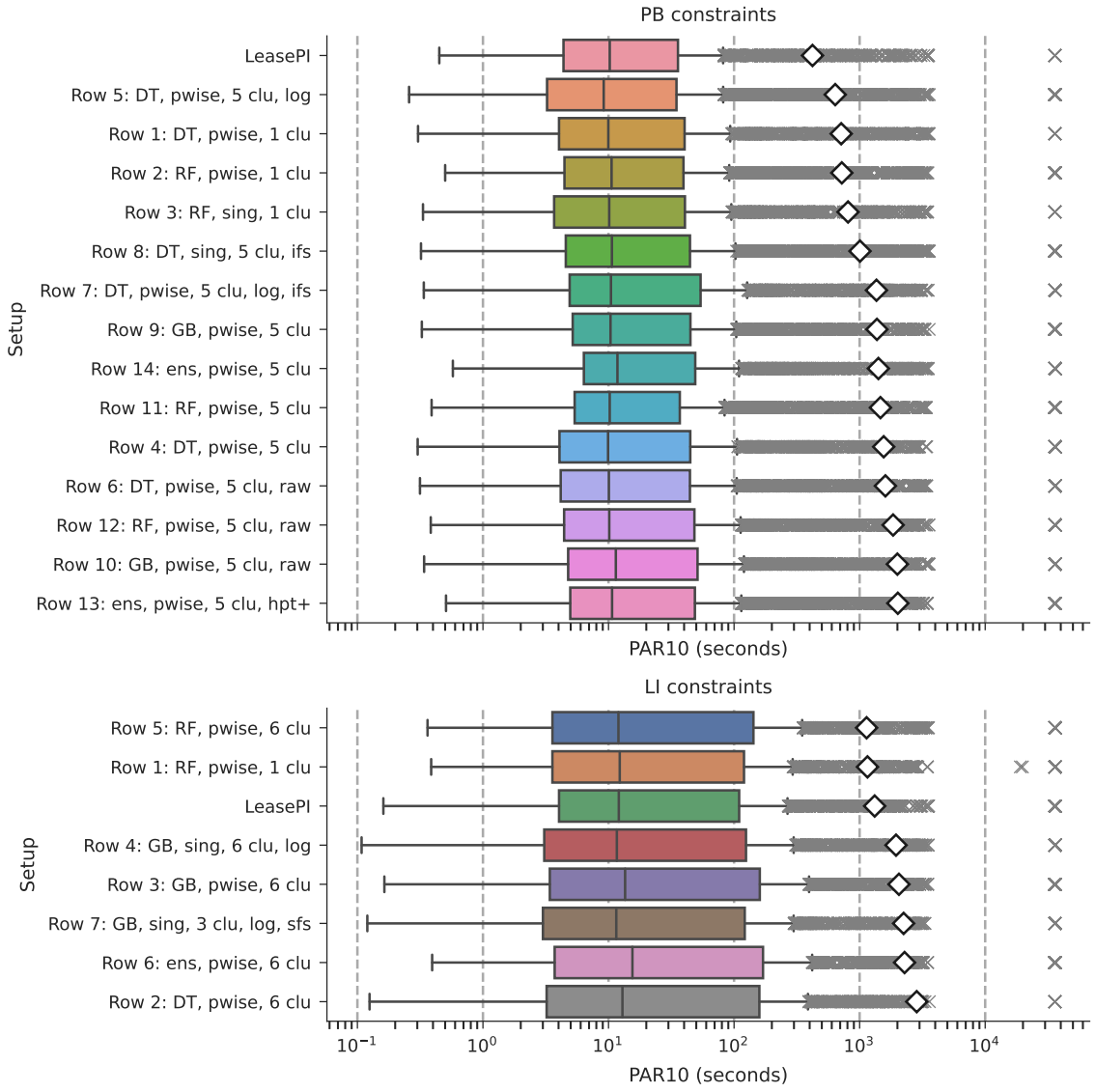


Figure 6.7: Distribution of PAR10 runtimes for various INDICON setups and LEASE-PI on a sample of instances. A sample of 10 results were taken for each instance from both the LEASE-PI experiments and the INDICON experiments. The setups are separated by constraint type, with PB at the top and LI at the bottom. The results ordered by the mean time, which is shown with a white diamond. The label for each setup refers to the rows in Table 6.5 and contains an abbreviated summary of each setup, as set out more fully in Table 6.5.

tion approach - we see that for PB problems, the fastest and slowest setups share every configuration detail apart from the fact that the fastest uses pairwise classifiers whereas the slowest uses a single 9-way classifier for the 9 available encodings (rows 2,3).

6.3.2 Comparison with LEASE-PI

In Figure 6.7 we compare the results of using INDICON at runtime to set the encodings with the simulated results from the best performing setup in LEASE-PI. This may not be a

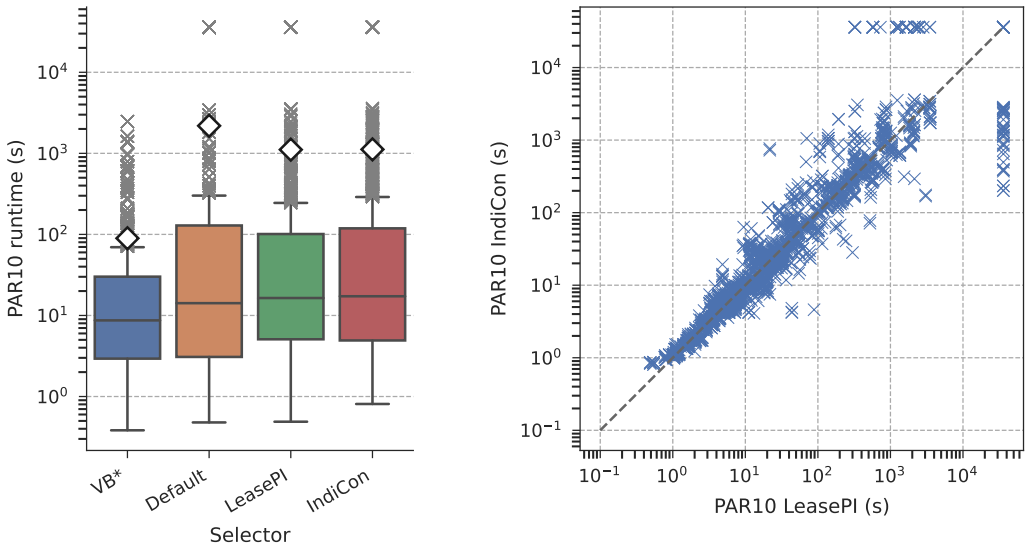


Figure 6.8: Comparison of IndiCon and LeaSE-PI performance on sample of instances when IndiCon has been used to set both PB and LI constraints. Left: distribution of PAR10 times for the single-choice virtual best (VB*), default encoding (Def), best LeaSE-PI setup and IndiCon with the best performing setups for each of PB and LI constraints. Right: the PAR10 times for the best LeaSE-PI setup versus IndiCon setting both types of encoding.

completely fair comparison given that LeaSE-PI was able to set both PB and LI constraints at the same time. What we can observe though is that the performance of the IndiCon setups is competitive.

Considering the PB experiments in Figure 6.7 we see that in terms of mean performance, none of the setups beat LeaSE-PI in this contest. The closest IndiCon setup by both mean and median is `dtree-cl2` which corresponds to row 5 in Table 6.5, with the next two setups matching the two top entries in the table, which also happen to be the best performers.

Looking at the setups which were predicting the LI encodings, the IndiCon performance is again similar to LeaSE-PI in terms of the median. This time, however, some of the setups do achieve a better mean PAR10, pushing LeaSE-PI into third place.

6.3.3 Setting Both PB and LI Constraints

The results of running Savile Row with IndiCon to set both PB and LI constraints is shown in Figure 6.8, once again comparing with the performance of LeaSE-PI. For this trial, Savile Row called on the best setups for each constraint type; these two setups are in bold in Table 6.5. Every solving run used a model which had not previously seen the problem class in question. The plots use the mean PAR10 time per instance when 10 results are sampled from each system (LeaSE-PI and IndiCon).

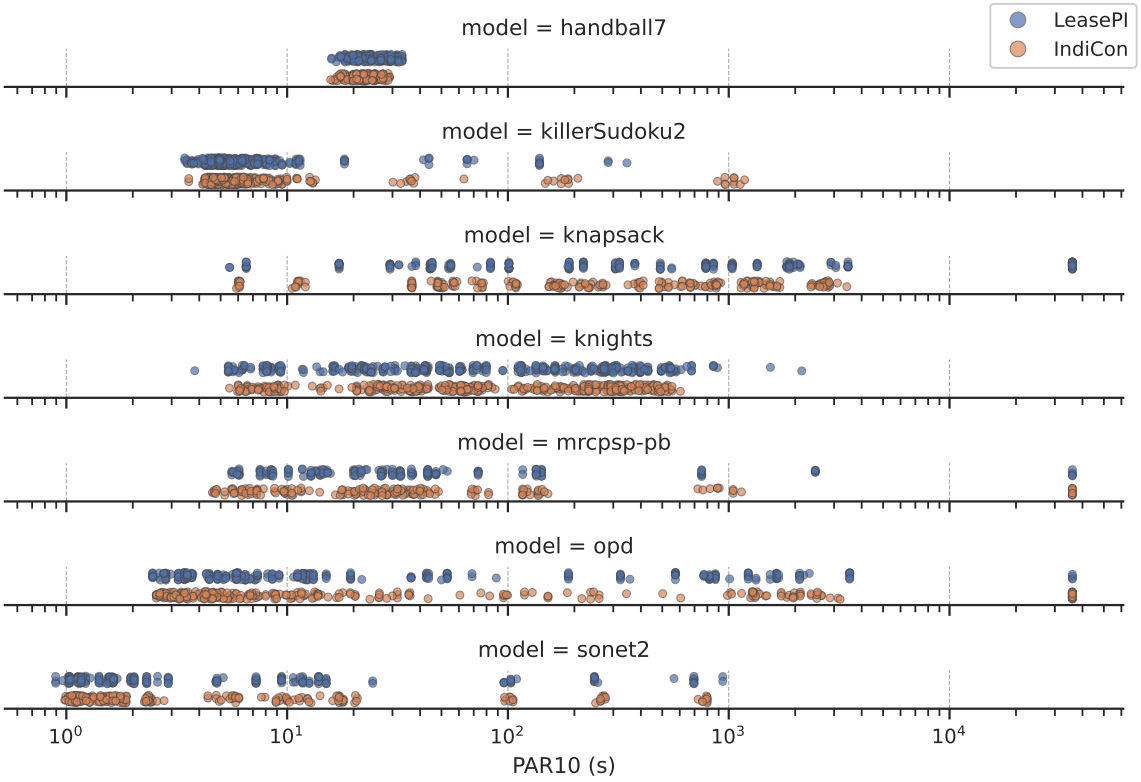


Figure 6.9: Comparison of IndiCon and LeaSE-PI performance broken down by constraint model where there were at least 10 instances in the sample.

The performance of IndiCon in this experiment very nearly matches the best LeaSE-PI setup. From the box plots we can see that the means are almost identical: 1118 seconds for LeaSE-PI and a slightly better 1111 seconds for IndiCon. For the rest of the distribution, LeaSE-PI has a slight edge. The better mean but worse median for IndiCon points to better performance in some of the harder problems.

From the scatter plot on the right of Figure 6.8 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. In terms of timeouts we see that both systems sometimes win and sometimes lose over each other.

One final comparison is shown in Figure 6.9, where we see a breakdown by constraint model (problem class). I have only plotted those problem classes for which at least 10 instances were available. Again we note the broadly similar performance of the two selectors. For two problem classes (`knapsack` and `knights`) IndiCon avoids some longer running times and timeouts suffered by LeaSE-PI. Conversely, LeaSE-PI does better on `killerSudoku2`.

6.4 Evaluation

6.4.1 Revisiting the Challenges

Back in Section 6.1 I explained some of the new challenges faced by trying to train and predict at the constraint level. It is perhaps worth reflecting on how successfully these challenges have been overcome by IndiCon and what work remains to be done.

1. I firstly pointed out that it was practically impossible to try out every encoding for every constraint in order to have a perfect training set. The compromise of clustering and varying one cluster's encodings in turn gave me the chance to try different encodings within problem instances.

2. To test IndiCon I had to actually implement an end-to-end system which made use of the ML-provided selections as part of the solving pipeline. This prototype had serious performance issues in terms of loading Python – the overhead time was discounted when calculating overall runtimes but clearly the pipeline would have to be optimised for IndiCon to be a practical solution for other users. What was definitely problematic was that it took a long time to try out new setups, especially if the changes were earlier on in the process, e.g. trying out another clustering strategy.

3. The implications of using the PB(AMO) encodings rather than the *Tree* encoding and of dealing with inequalities resulting from an equality constraint were overcome with relative ease by extending Savile Row.

6.4.2 Back to MRCPSP

At the start of the chapter I presented a motivating example focussed on the MRCPSP scheduling problem. By inspecting the constraint model, it seemed like the PB constraints could be intuitively split into two groups. Having carried out clustering on the whole corpus of problems involving PB constraints, I thought it would be interesting to see whether the intuition was reflected by the automated categorisation.

Following the results of the agglomerative clustering, as described in Section 6.2.3, I chose to use 5 clusters for all the PBs represented in the corpus. In the corpus were 20 instances of MRCPSP. Of these, the clustering method assigned a single cluster to 2 of the instances, three clusters to one further instance and two clusters to the remaining majority of 17 instances. It's not clear whether the two clusters identified by the unsupervised

algorithm in most instances are split in the same way as the difference we observed in the model, but it is encouraging to see that there is broad agreement about there being two groups of constraints with different properties.

6.4.3 Conclusions

IndiCon was a natural next step after designing, investigating and reporting on the selection of encodings for entire instances (LeaSE-PI). The work on IndiCon is exploratory and just begins to address some of the challenges and to draw out some lessons. I attempt below to distil what I have observed through the work and the results presented in this chapter.

There is potential for improving on LeaSE-PI There are many cases where IndiCon outperformed LeaSE-PI on individual problems. With more investigation it may be possible to beat LeaSE-PI more consistently.

The configuration space is barely explored I showed that there are many design choices in the setup of IndiCon and I have only been able to investigate a few combinations.

There was a lot of wasted time Once clusters were assigned and systematic encoding selections were set up, these were used to solve the problems in the corpus. However, it may have been possible to stop early and abandon solving runs which were not going to beat the current best for a particular cluster. A more dynamic approach might enable further investigations of the configuration space in the same amount of compute time.

Simple seems best In several aspects, it seems that the simpler solution led to better results. The most glaring example is that it was possible to get better results when using unclustered, single-choice data to obtain the training set, as observed in the best performing setup for PB constraints. Similarly, the use of a custom loss function for the hyperparameter tuning did not generally yield better predictions. Even the choice of classifier points to the possibility that simplest is best, with basic decision trees performing close to the best, at least for PB constraints.

IndiCon could offer better modularity I essentially carried out the same investigation separately for PB and LI constraints, sharing the approaches and techniques, but separating out the data. At the end, however, it was possible for Savile Row to call on IndiCon for two different types of constraints and use their suggestions in the same

solving run. This idea is open to expansion, where trained models can be learned for any types of constraints for which SAT encodings with complementary strengths exist.

IndiCon should scale better The preferred setup in LeaSE-PI, which was the best performing in the split-by-class experiment, would struggle to scale easily beyond two constraint types. We discussed in the previous chapter how the ML setup which predicted the PB and LI encodings separately might be the foundation of a more scalable solution based on LeaSE-PI. Similarly IndiCon would not suffer from being appliedto more dimensions in configuration space, i.e. more constraint types.

6.5 Summary

In this chapter we broke the problem of chosing SAT encodings down to the level of individual constraints, investigating whether we can take advantage of this finer-grained control to improve performance further. I began with the intuition that when looking at some problems written in a constraint modelling language, it is possible to see that for a given constraint type, different constraints are introduced with potentially different structures.

I described a few challenges that needed to be overcome in order to train models and test the results meaningfully. I gave a detailed account of the method used to implement IndiCon, my name for a system for learning to select SAT encodings at the level of individual constraints.

IndiCon was tested thoroughly and compared to the predecessor described in earlier chapters, which I named LeaSE-PI for convenience. A variety of tests were presented to help analyse the effectiveness of this approach.

I showed that IndiCon is competitive with LeaSE-PI and outperforms it in some cases. I also noted some suprising findings about what seemed to work well, which often tended to be the simpler design solutions. The investigation into IndiCon is limited, not least by the fact that trying out different setups is very time-consuming. However, there is potential to use IndiCon as a key component of a scalable, modular system for selecting good SAT encodings across a range of constraint types.

Chapter 7

Conclusion

♫ I'm sure everything will end up alright ♫

Audioslave

At the start of this document, I set out the proposed thesis and some more specific research questions which this dissertation has attempted to address. In this chapter I:

- summarise the findings from the three technical chapters in light of the thesis statement and research questions,
- reflect on some of the limitations of this research, and
- propose potential future work based on remaining open questions.

7.1 Revisiting the Thesis and Research Questions

The thesis was initially posed as follows:

Thesis

The solving of constraint satisfaction problems can be made faster by using machine learning to select SAT encodings for constraints.

The work presented in this dissertation supports the thesis as far as selecting SAT encodings for pseudo-Boolean (PB) and linear integer (LI) constraints for problems expressed in the Essence Prime.

The three technical chapters each support the assertion as they approach the challenge with increasing levels of specialisation. In Chapter 4 generic instance features are

used to select encodings with some success, reducing the gap between the single best and the virtual best encodings. When constraint-specific features are used in Chapter 5 we are able to close the gap even further, especially for unseen problem classes. Finally, in Chapter 6 we see that in some cases we can improve the performance further by setting the encodings for each constraint in an instance separately.

Let us review the findings in more detail, using the initial research questions as prompts.

Research Question 1

To what extent does the choice of SAT encoding for constraints affect problem-solving performance?

The literature we review in the background chapters points to the potential performance gains if we can choose SAT encodings well. In [43] the authors report large variations in encoding size and solving time between their eight PB(AMO) encodings, with the best encoding varying between different problem classes and indeed for different instances within a problem class. The MeSAT [81] and Proteus [82] systems were both shown to benefit from choosing different variable encodings by using machine learning (ML) on features of problem instances. The PB and LI encodings implemented in Savile Row use different variable encodings too, so we also expect to make gains in our context.

This expectation is indeed confirmed by the investigation into a complementary portfolio of encodings presented in Chapter 4. We see that if we simply choose the one combination of encodings for PB and LI constraints which is the best overall, we solve the corpus over 10 times slower than if we had access to the full range of encodings (see Figure 4.1). The next challenge was therefore to unlock this complementarity by using ML to predict the best encodings.

Research Question 2

Can a good SAT encoding choice be made for unseen CSP instances based on generic instance features?

In Chapter 4 I describe the LeaSE-PI framework which implements a number of ML setups to choose between a subset of available encodings. This subset (or portfolio) is constructed by determining the best 6 combinations of encodings for PB and LI constraints as observed from data in the training set. The portfolio size is a practical compromise between training time and complementarity.

The instances are initially represented by "off-the-shelf" generic features generated

by the FZN2FEAT tool [85] – these features only contain general information about the variables and their domains, as well as about the types of global constraints present in the instance. Even so, we are able to make predictions which lead to a 75% reduction in PAR10 time compared to the single best (SB) on seen problem classes and a 45% reduction for unseen problem classes.

Chapter 4 also comments on the toolkit of available encodings from which we're selecting. We observe that the default encoding provided by SAVILE ROW (*Tree*) is very competitive, being the best choice both for PB and LI most often for our corpus of problems. However, that does not necessarily mean it's the best encoding overall; in fact the GGPW encoding is the best if the goal is to bring the overall runtime down for solving the entire corpus.

Research Question 3

Can the quality of encoding selection be improved by the use of features which are specific to the relevant constraints?

This question too was positively answered in Chapter 5, where I construct a new featureset *lipb* based entirely on the constraints we are interested in. Not only do these extra features help when combined with the generic instance features, the specialised featureset on its own is actually sufficient to match the performance of the combined featureset in many of the ML setups. When it comes to predicting encodings for unseen problem classes, the *lipb* features lead to the best performance, in particular when used with random forests in a pairwise classification setup. In this setting, the time saved compared to single best (SB) increases to 57%.

In the same chapter we compare the performance of the ML setup constructed in this paper with AUTOFOLIO, which is an advanced algorithm selection and configuration system. The solution presented here, LEASE-PI, is approximately twice as fast as AUTOFOLIO when predicting for unseen problem classes and five times faster on known problem classes. It is interesting to note that AUTOFOLIO performed best when using just the *lipb* featureset.

With the introduction of new features I also study feature importance in our context. It is very difficult to isolate the importance of an individual feature, but we are able to make some general observations about the featuresets in this study. For example, the top 20 most important features contain a roughly equal balance of generic and specialised features when predicting for known problem classes; however, when predicting for unseen classes, the specialised PB/LI features are a lot more valuable.

Research Question 4

Is it practical to learn to set encodings for individual constraints within a problem instance? Does it lead to performance improvements compared to a single encoding choice per constraint type?

I introduce IndiCon in Chapter 6, in which we learn that selecting SAT encodings for individual constraints can work well and certainly outperforms the previous LeaSE-PI approach for some problems. However, the overall performance across the entire problem corpus is in fact very similar, so one might question whether the extra complication is worth it. With so many design choices to be made, the scope for exploration is vast. I set out many aspects which can be varied, such as: how to group constraints for obtaining training data, the use of whole-instance features for added context, a variety of ML algorithms, different custom scores for hyperparameter tuning, just to name a few. Even with the limitations in computing resource and time, it was possible to "win" on many instances with some of these setups, so I believe the potential remains for much more significant gains across a wider range of problems.

During the IndiCon investigation it emerged that simpler is best on at least two occasions. Firstly decision trees were able to get fairly close to the performance of random forests. This is exciting because decision trees are explainable and could potentially be used to construct a sensible default heuristic for choosing encodings per-constraint. Secondly, the elaborate scheme I created for obtaining clustered training data was in fact outperformed by the training set in which I used 1 cluster, in other words where I labelled each constraint simply based on what worked well for the parent instance as a whole.

In summary, the answer to the research question is a tentative yes: IndiCon is a solution which sets encodings separately for each constraint within Savile Row, and it does lead to improved performance over LeaSE-PI, but only slightly and with great variation across problem classes. However, with so many design choices available, and just a few explored within the scope of this thesis, more significant gains may be possible.

7.2 Lessons Learned

I believe some overarching conclusions may be drawn from the work presented in this thesis dissertation:

Good defaults are possible with ML In the introduction I mention the "holy grail" of programming where a user states the problem and the computer solves it. In that spirit, this thesis has shown that it is possible to build a constraint solving pipeline which helps non-expert users to choose sensible initial settings (in our case SAT encodings). This is surely a good ambition. Of course a constraint programming (CP) expert employed to tackle a specific problem type for an industrial application can still have their own say over encodings, but the settings obtained by ML could provide a good starting place for such an advanced user to begin fine tuning.

Clarity of Purpose is Important ML literature provides many ways to measure the success of a particular trained model – in the case of a classification problem such as ours, that metric might simply be the percentage accuracy, i.e. what proportion of the model's predictions are exactly the same as the very best? We've seen that as an overall metric this is quite brittle, especially in our case, where second-best might well be good enough. It is important therefore to be clear about the overall purpose or objective. I aimed to solve a set of problems as quickly as possible. We see that the most popular encoding configuration across the corpus is not necessarily the choice which solves the entire corpus in the quickest time. I used this reasoning to try to improve the performance, for instance by adding a custom scoring function to the hyperparameter tuning stage.

7.3 Limitations

Hopefully the findings summarised above can inform practice and inspire further research. I re-state below some of the known limitations of this work, so that the findings may be understood in their proper context.

Constraint Types The thesis explores only PB and LI constraints. This is both because they can make a big difference to solving time and because a large number of complementary encodings are available in Savile Row.

Problem Corpus The intention was to use as wide a variety of problems as possible, but to be able to use Savile Row; I was therefore limited to problems which were already expressed in Essence Prime, or could be translated faithfully and easily from other constraint languages. This means that the corpus is still quite limited and could be expanded much further by a dedicated effort to translate more constraint models from the more popu-

lar MiniZinc language [25], or by carrying out a similar work using a different constraint solving pipeline.

Problem Size For practical reasons I set a runtime limit of 1 hour for my investigations. Some more challenging problems exist which would take longer to solve and for which the choice of SAT encoding could be crucial.

7.4 Future Work

More Constraint Types One easy extension of this work is to consider other constraint types where a suite of encodings with complementary strengths is available. For instance I informally carried out some experiments with the at-most-one (AMO) constraint, for which several SAT encodings are available in Savile Row – this trial showed that different encodings were best on different problems.

Constraint Neighbourhood Constraints in constraint satisfaction problems (CSPs) do not tend to occur in isolation – there is often overlap between the variables involved in different constraints. It might be informative to gather constraint features about neighbouring constraints, especially in the case where we're making selections per-constraint.

Graph Neural Networks Constraints problems are often represented as graphs. Graph neural network (GNN)s have received much attention recently and have been applied in various ways to the field of constrained optimisation (CO) [95]. One possible research direction would be to use GNNs to predict encodings per constraint or per instance. The graph representation may be able to capture the structure of a problem better than tabular data.

More Efficient Generation of Training Data The generation of timing data was particularly time-consuming in IndiCon especially when trying out different encoding assignments for the same instance. It may be possible to dramatically reduce the number of wasted runs by employing something like the capping method in irace [96] to stop a solving run which has exceeded the best time already obtained.

Heuristics from Simple ML Models In IndiCon decision trees proved quite competitive with random forests. By inspecting the learned trees, it may be possible to design good

simple heuristics for constraint tools such as Savile Row to choose how to encode a constraint on-the-fly.

Appendices

Appendix A

Featuresets

I use four featuresets to make per-instance predictions in this thesis, as described in the technical chapters (Chapters 4 to 6). This appendix lists the features in the F2F featureset, which is provided by the `fzn2feat` tool [85]. The other featuresets are not described here because:

- F2FSR attempts to extract the same features as F2F
- LIPB is described in detail in the dissertation
- COMBI is simply the union of F2FSR and LIPB.

Table A.1: Name and brief description of each feature in the F2F featureset as produced by `fzn2feat`

CONSTRAINT FEATURES (27)	
`c_avg_deg_cons`	Average of the constraints degree
`c_avg_dom_cons`	Average of the constraints domain
`c_avg_domdeg_cons`	Average of the ratio constraints domain/degree
`c_bounds_d`	No. of constraints using "boundsD" annotation
`c_bounds_r`	No. of constraints using "boundsR" annotation
`c_bounds_z`	No. of constraints using "boundsZ" or "bounds" annotation
`c_cv_deg_cons`	Coefficient of Variation of constraints degree
`c_cv_dom_cons`	Coefficient of Variation of constraints degree
`c_cv_domdeg_cons`	Coefficient of Variation of the ratio constraints domain/degree

c_domain	No. of constraints using "domain" annotation
c_ent_deg_cons	Entropy of constraints degree
c_ent_dom_cons	Entropy of constraints domain
c_ent_domdeg_cons	Entropy of the ratio constraints domain/degree
c_logprod_deg_cons	Logarithm of the product of constraints degree
c_logprod_dom_cons	Logarithm of the product of constraints domain
c_max_deg_cons	Maximum of the constraints degree
c_max_dom_cons	Maximum of the constraints domain
c_max_domdeg_cons	Maximum of the ratio constraints domain/degree
c_min_deg_cons	Minimum of the constraints degree
c_min_dom_cons	Minimum of the constraints domain
c_min_domdeg_cons	Minimum of the ratio constraints domain/degree
c_num_cons	Total no. of constraints
c_priority	No. of constraints using "priority" annotation
c_ratio_cons	Ratio no. of constraints / no. of variables
c_sum_ari_cons	Sum of constraints arity
c_sum_dom_cons	Sum of constraints domain
c_sum_domdeg_cons	Sum of the ratio constraints domain/degree
DOMAIN FEATURES (18)	
d_array_cons	No. of array constraints
d_bool_cons	No. of boolean constraints
d_bool_vars	No. of boolean variables
d_float_cons	No. of float constraints
d_float_vars	No. of float variables
d_int_cons	No. of integer constraints
d_int_vars	No. of integer variables
d_ratio_array_cons	Ratio array constraints / total no. of constraints
d_ratio_bool_cons	Ratio boolean constraints / total no. of constraints
d_ratio_bool_vars	Ratio boolean variables / total no. of variables
d_ratio_float_cons	Ratio float constraints / total no. of constraints
d_ratio_float_vars	Ratio float variables / total no. of variables

d_ratio_int_cons	Ratio integer constraints / total no. of constraints
d_ratio_int_vars	Ratio integer variables / total no. of variables
d_ratio_set_cons	Ratio set constraints / total no. of constraints
d_ratio_set_vars	Ratio set variables / total no. of variables
d_set_cons	No. of set constraints
d_set_vars	No. of set variables
GLOBAL CONSTRAINT FEATURES (4)	
gc_diff_globs	No. of different global constraints
gc_global_cons	Total no. of global constraints
gc_ratio_diff	Ratio different global constraints / no. of global constraints
gc_ratio_globs	Ratio no. of global constraints / total no. of constraints
OBJECTIVE FEATURES (8)	
o_deg	Degree of the objective variable
o_deg_avg	Ratio degree of the objective variable / average of var. degree
o_deg_cons	Ratio degree of the objective variable / number of constraints
o_deg_std	Standardization of the degree of the objective variable
o_dom	Domain size of the objective variable
o_dom_avg	Ratio domain of the objective variable / average of var. domain
o_dom_deg	Ratio domain of the objective variable / degree of the obj. var
o_dom_std	Standardization of the domain of the objective variable
SOLVING FEATURES (11)	
s_bool_search	Number of "bool_search" annotations
s_first_fail	Number of "int_search" annotations
s_goal	Solve goal (1 = satisfy, 2 = minimize, 3 = maximize)
s_indomain_max	Number of "indomain_max" annotations
s_indomain_min	Number of "indomain_min" annotations
s_input_order	Number of "input_order" annotations
s_int_search	Number of "int_search" annotations
s_labeled_vars	Number of variables to be assigned
s_other_val	Number of other value search heuristics

s_other_var	Number of other variable search heuristics
s_set_search	Number of "set_search" annotations

VARIABLE FEATURES (27)

v_avg_deg_vars	Average of the variables degree
v_avg_dom_vars	Average of the variables domain
v_avg_domdeg_vars	Average of the ratio variables domain/degree
v_cv_deg_vars	Coefficient of Variation of variables degree
v_cv_dom_vars	Coefficient of Variation of variables degree
v_cv_domdeg_vars	Coefficient of Variation of the ratio variables domain/degree
v_def_vars	Number of defined variables
v_ent_deg_vars	Entropy of variables degree
v_ent_dom_vars	Entropy of variables domain
v_ent_domdeg_vars	Entropy of the ratio variables domain/degree
v_intro_vars	Number of introduced variables
v_logprod_deg_vars	Logarithm of the product of variables degree
v_logprod_dom_vars	Logarithm of the product of variables domain
v_max_deg_vars	Maximum of the variables degree
v_max_dom_vars	Maximum of the variables domain
v_max_domdeg_vars	Maximum of the ratio variables domain/degree
v_min_deg_vars	Minimum of the variables degree
v_min_dom_vars	Minimum of the variables domain
v_min_domdeg_vars	Minimum of the ratio variables domain/degree
v_num_aliases	Number of alias variables
v_num_consts	Number of constant variables
v_num_vars	Total no. of variables variables
v_ratio_bounded	Ratio (aliases + constants) / total no. of variables
v_ratio_vars	Ratio no. of variables / no. of constraints
v_sum_deg_vars	Sum of variables degree
v_sum_dom_vars	Sum of variables domain
v_sum_domdeg_vars	Sum of the ratio variables domain/degree

Appendix B

Neural Network Trial

In early experiments I included a neural network in the set of ML models used to predict good encodings. I used `MLPClassifier` from the `scikit-learn` library [6]: a basic feed-forward neural network which trains using backpropagation. I left many choices to the default, such as using the *ReLU* activation function, and the *adam* optimiser. As with other models, I used `RandomizedSearchCV` to tune some hyperparameters including the hidden layer structure.

I show here the results of one trial as an example. I used the *lipb* and *combi* featuresets to train and predict using the *split-by-class* policy. For the interested reader, Table B.2 shows the resulting hyperparameter choices from one iteration of training. The performace results in terms of PAR10 time compared to the virtual best over 50 split/train/predict cycles are shown in Table B.1. We can see that the prediction quality is not as good as when we use random forests. It also takes considerably longer to train the neural network classifier.

Table B.1: A limited set of results from a trial of MLP-based classifiers in the split-by-class scenario, using two featuresets. Performance is measured as the mean PAR10 time divided by the mean virtual best time. The table shows the best performing setup from LeaSE-PI too for comparison. The training times are approximate times as the test was run on a laptop - the "user" time is given in minutes.

		Results ($\times$ VB)	
Selector	Training time	*lipb*	*combi*
Pairwise RF	21 CPU minutes	11.00	11.84
MLP	86 CPU minutes	13.73	14.34

Table B.2: Best hyperparameter values found by randomised search with cross-validation on one training run using the split-by-class policy and the *lipb* features. The first column shows which encoding configurations were being paired up. The tuned hyperparameters are the hidden layer structure, whether early stopping is used if the loss is not dropping any more, and the L2 regularisation rate.

	Hyperparameters		
Encodings pair	Hidden Layer Sizes	Early Stopping?	L2 Rate (α)
`tree_mdd/gpw_ggth`	[230, 72, 34]	True	0.042
`tree_mdd/rggt_tree`	[217, 164, 99]	True	0.095
`tree_mdd/ggth_mdd`	[148, 122, 101]	True	0.089
`tree_mdd/gpw_ggt`	[245, 240, 165, 27]	False	0.084
`tree_mdd/gpw_swc`	[244, 214, 208, 199, 164]	True	0.037
`gpw_ggth/rggt_tree`	[165, 114, 109, 62, 47]	False	0.058
`gpw_ggth/ggth_mdd`	[195, 185, 127, 33]	False	0.100
`gpw_ggth/gpw_ggt`	[194, 75, 23]	True	0.084
`gpw_ggth/gpw_swc`	[217, 164, 99]	True	0.095
`rggt_tree/ggth_mdd`	[230, 72, 34]	True	0.042
`rggt_tree/gpw_ggt`	[134, 86, 81]	False	0.026
`rggt_tree/gpw_swc`	[194, 75, 23]	True	0.084
`ggth_mdd/gpw_ggt`	[195, 185, 127, 33]	False	0.100
`ggth_mdd/gpw_swc`	[74, 18]	True	0.095
`gpw_ggt/gpw_swc`	[230, 72, 34]	True	0.042

Appendix C

Research Data Statement

As part of preparing the journal paper [3] which covers the work in Chapters 4 and 5 I set up a public repository at https://github.com/felixvuo/lease-data where interested readers can access the software, problem corpus and results. It is also available under a DOI at https://doi.org/10.5281/zenodo.8380691.

Bibliography

[1] Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Selecting SAT Encodings for Pseudo-Boolean and Linear Constraints: Preliminary Results, October 2021.

[2] Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Selecting SAT Encodings for Pseudo-Boolean and Linear Integer Constraints. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.38.

[3] 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.

[4] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.

[5] 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.

[6] 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.

[7] Michael L. Waskom. Seaborn: Statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. doi:10.21105/joss.03021.

[8] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[9] Özgür Akgün, Alan M. Frisch, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic Generation of Constraint Models from Problem Specifications. *Artificial Intelligence*, 310:103751, September 2022. doi:10.1016/j.artint.2022.103751.

[10] Peter Nightingale. Savile Row Manual, November 2021. arXiv:2201.03472, doi:10.48550/arXiv.2201.03472.

[11] Andreas Drexl and Juergen Gruenewald. Nonpreemptive Multi-Mode Resource-Constrained Project Scheduling. *IIE Transactions*, 25(5):74–81, September 1993. doi:10.1080/07408179308964317.

[12] Sameela Suharshani Wijesundara, Maria Garcia de la Banda, and Guido Tack. Addressing problem drift in UNHCR fund allocation. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.37.

[13] Younes Aalian, Gilles Pesant, and Michel Gamache. Optimization of short-term underground mine planning using constraint programming. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.6.

[14] Vincent Barbosa Vaz, James Bailey, Christopher Leckie, and Peter J. Stuckey. Predict-Then-Optimise Strategies for Water Flow Control. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:10, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2023.42.

[15] Eugene C. Freuder. Progress towards the Holy Grail. *Constraints*, 23(2):158–171, April 2018. doi:10.1007/s10601-017-9275-0.

[16] Hongbo Li, Yaling Wu, Minghao Yin, and Zhanshan Li. A Portfolio-Based Approach to Select Efficient Variable Ordering Heuristics for Constraint Satisfaction Problems. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:10, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.32.

[17] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed. https://github.com/chuffed/chuffed/, 2018.

[18] Christian Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, January 1994. doi:10.1016/0004-3702(94)90041-8.

[19] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-74970-7.

[20] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, September 2008. doi:10.1007/s10601-008-9047-y.

[21] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva Del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102. IOS Press, 2006.

[22] Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Automatically Exploiting Subproblem Equivalence in Constraint Programming. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 71–86, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-13520-0_10.

[23] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a Lazy Clause Generation Solver. https://github.com/chuffed/chuffed/, 2018.

[24] Gecode Team. Gecode: Generic constraint development environment. https://www.gecode.org/, 2019.

[25] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The MiniZinc Challenge 2008–2013. *AI Magazine*, 35(2):55–60, June 2014. doi:10.1609/aimag.v35i2.2539.

[26] Laurent Perron and Vincent Furnon. OR-Tools. https://developers.google.com/optimization/, March 2022.

[27] MiniZinc Challenge 2022. https://www.minizinc.org/challenge2022/challenge.html.

[28] Neng-Fa Zhou and Håkan Kjellerstrand. The Picat-SAT Compiler. In Marco Gavanelli and John Reppy, editors, *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, pages 48–62, Cham, 2016. Springer International Publishing. doi:10.1007/978-3-319-28228-2_4.

[29] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series b*, pages 51–53. University of Helsinki, 2020.

[30] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems, November 2022. arXiv:1611.03398, doi:10.48550/arXiv.1611.03398.

[31] Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2022 XCSP3 Competition, September 2022. arXiv:2209.00917.

[32] SAT Competitions. https://satcompetition.github.io/.

[33] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. doi:10.1145/368273.368557.

[34] Armin Biere. SAT: CPAIOR 2020 Masterclass. http://fmv.jku.at/biere/talks/Biere-CPAIOR20-tutorial.pdf, September 2020.

[35] Armin Biere. SAT: CPAIOR 2020 Masterclass. https://www.youtube.com/watch?v=t3bJcU1aQJI, September 2020.

[36] JP Marques Silva and KA Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227. IEEE, 1996. doi:10.1109/ICCAD.1996.569607.

[37] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Chapter 4. Conflict-Driven Clause Learning SAT Solvers. *Handbook of Satisfiability*, pages 133–182, 2021. doi:10.3233/FAIA200987.

[38] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[39] Hendrik Bierlee, Graeme Gange, Guido Tack, Jip J. Dekker, and Peter J. Stuckey. Coupling Different Integer Encodings for SAT. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 44–63, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-08011-1_5.

[40] Ian P. Gent. Arc consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[41] 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*, pages 181–194, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[42] Christian Bessiere, George Katsirelos, Nina Narodytska, and Toby Walsh. Circuit Complexity and Decompositions of Global Constraints, May 2009. arXiv:0905.3757, doi:10.48550/arXiv.0905.3757.

[43] 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.

[44] Tobias Philipp and Peter Steinke. PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, Lecture Notes in Computer Science, pages 9–16, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24318-4_2.

[45] 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.

[46] 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, IJCAI-17*, pages 555–562, 2017. doi:10.24963/ijcai.2017/78.

[47] 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.

[48] 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.

[49] Aolong Zha, Miyuki Koshimura, and Hiroshi Fujita. N-level Modulo-Based CNF encodings of Pseudo-Boolean constraints for MaxSAT. *Constraints*, 24(2):133–161, April 2019. doi:10.1007/s10601-018-9299-0.

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

[51] 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.

[52] 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.

[53] Jamie Miles, Janette Turner, Richard Jacques, Julia Williams, and Suzanne Mason. Using machine-learning risk prediction models to triage the acuity of undifferentiated patients entering the emergency care system: A systematic review. *Diagnostic and Prognostic Research*, 4(1):16, October 2020. doi:10.1186/s41512-020-00084-1.

[54] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, April 2021. doi:10.1016/j.ejor.2020.07.063.

[55] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. doi:10.1038/nature14539.

[56] Jesper E. van Engelen and Holger H. Hoos. A survey on semi-supervised learning. *Machine Learning*, 109(2):373–440, February 2020. doi:10.1007/s10994-019-05855-6.

[57] Longlong Jing and Yingli Tian. Self-Supervised Visual Feature Learning With Deep Neural Networks: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(11):4037–4058, November 2021. doi:10.1109/TPAMI.2020.2992393.

[58] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Li Mian, Zhaoyu Wang, Jing Zhang, and Jie Tang. Self-Supervised Learning: Generative or Contrastive. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):857–876, January 2023. doi:10.1109/TKDE.2021.3090866.

[59] Leo Breiman. *Classification and Regression Trees*. Routledge, New York, October 2017. doi:10.1201/9781315139470.

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

[61] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, April 2006. doi:10.1007/s10994-006-6226-1.

[62] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, February 2002. doi:10.1016/S0167-9473(01)00065-2.

[63] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, San Francisco California USA, August 2016. ACM. doi:10.1145/2939672.2939785.

[64] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995. doi:10.1007/BF00994018.

[65] Rob Ashmore, Radu Calinescu, and Colin Paterson. Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges. *ACM Computing Surveys*, 54(5):111:1–111:39, May 2021. doi:10.1145/3453444.

[66] F. J. Ferri, P. Pudil, M. Hatef, and J. Kittler. Comparative study of techniques for large-scale feature selection. In Edzard S. Gelsema and Laveen S. Kanal, editors, *Machine Intelligence and Pattern Recognition*, volume 16 of *Pattern Recognition in Practice IV*, pages 403–413. North-Holland, January 1994. doi:10.1016/B978-0-444-81892-8.50040-7.

[67] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *Proceedings of the 30th International Conference on Machine Learning*, pages 1238–1246. PMLR, May 2013.

[68] Roberto Confalonieri, Ludovik Coba, Benedikt Wagner, and Tarek R. Besold. A historical perspective of explainable Artificial Intelligence. *WIREs Data Mining and Knowledge Discovery*, 11(1):e1391, 2021. doi:10.1002/widm.1391.

[69] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-End Constrained Optimization Learning: A Survey, March 2021. arXiv:2103.16378, doi:10.48550/arXiv.2103.16378.

[70] Natalia Vesselinova, Rebecca Steinert, Daniel F. Perez-Ramirez, and Magnus Boman. Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking. *IEEE Access*, 8:120388–120416, 2020. doi:10.1109/ACCESS.2020.3004964.

[71] Daphné Lafleur, Sarath Chandar, and Gilles Pesant. Combining Reinforcement Learning and Constraint Programming for Sequence-Generation Tasks with Hard Constraints. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.30.

[72] John R. Rice. The Algorithm Selection Problem. In Morris Rubinoff and Marshall C. Yovits, editors, *Advances in Computers*, volume 15, pages 65–118. Elsevier, January 1976. doi:10.1016/S0065-2458(08)60520-3.

[73] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25566-3_40.

[74] Mario Andrés Muñoz, Hamed Soleimani, and Sevvandi Kandanaarachchi. Benchmarking algorithm portfolio construction methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '22, pages 499–502, New York, NY, USA, July 2022. Association for Computing Machinery. doi:10.1145/3520304.3528880.

[75] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, August 2016. doi:10.1016/j.artint.2016.04.003.

[76] Nguyen Dang. A portfolio-based analysis method for competition results. In *ModRef 2022*, Haifa, Israel, July 2022.

[77] Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of Green AI. *WIREs Data Mining and Knowledge Discovery*, 13(4):e1507, 2023. doi:10.1002/widm.1507.

[78] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi, editors, *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*, pages 149–190. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-50137-6.

[79] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008. doi:10.1613/jair.2490.

[80] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: A sequential CP portfolio solver. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1861–1867, Salamanca, Spain, April 2015. Association for Computing Machinery. doi:10.1145/2695664.2695741.

[81] 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.

[82] 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.

[83] 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.

[84] Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. XCSP3 and its ecosystem. *Constraints*, February 2020. doi:10.1007/s10601-019-09307-9.

[85] 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.

[86] 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.

[87] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[88] 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.

[89] 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. 2021.

[90] András Vargha and Harold D. Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, June 2000. doi:10.3102/10769986025002101.

[91] Geoffrey Neumann, Mark Harman, and Simon Poulding. Transformed Vargha-Delaney Effect Size. In Márcio Barros and Yvan Labiche, editors, *Search-Based Software Engineering*, Lecture Notes in Computer Science, pages 318–324, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-22183-0_29.

[92] Alfredo S. Ramos, Pablo A. Miranda-Gonzalez, Samuel Nucamendi-Guillén, and Elias Olivares-Benitez. A Formulation for the Stochastic Multi-Mode Resource-Constrained Project Scheduling Problem Solved with a Multi-Start Iterated Local Search Metaheuristic. *Mathematics*, 11(2):337, January 2023. doi:10.3390/math11020337.

[93] Rainer Kolisch and Arno Sprecher. PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research*, 96(1):205–216, January 1997. doi:10.1016/S0377-2217(96)00170-1.

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

[95] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial Optimization and Reasoning with Graph Neural Networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

[96] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, January 2016. doi:10.1016/j.orp.2016.09.002.