Automatic Feature Learning for Essence: a Case Study on Car Sequencing

Alessio Pellegrino
Dept. of Computer Science and Engineering, University of Bologna, Italy

Özgür Akgün
School of Computer Science, University of St Andrews, Scotland

Nguyen Dang
School of Computer Science, University of St Andrews, Scotland

Zeynep Kiziltan
Dept. of Computer Science and Engineering, University of Bologna, Italy

Ian Miguel
School of Computer Science, University of St Andrews, Scotland

Abstract

Constraint modelling languages such as ESSENCE offer a means to describe combinatorial problems at a high-level, i.e., without committing to detailed modelling decisions for a particular solver or solving paradigm. Given a problem description written in ESSENCE, there are multiple ways to translate it to a low-level constraint model. Choosing the right combination of a low-level constraint model and a target constraint solver can have significant impact on the effectiveness of the solving process. Furthermore, the choice of the best combination of constraint model and solver can be instance-dependent, i.e., there may not exist a single combination that works best for all instances of the same problem. In this paper, we consider the task of building machine learning models to automatically select the best combination for a problem instance. A critical part of the learning process is to define *instance features*, which serve as input to the selection model. Our contribution is automatic learning of instance features directly from the high-level representation of a problem instance using a language model. We evaluate the performance of our approach using the ESSENCE modelling language with a case study involving the car sequencing problem.

2012 ACM Subject Classification Computing methodologies → Artificial intelligence

Keywords and phrases Constraint modelling, algorithm selection, feature extraction, machine learning, language model

1 Introduction

In many domains, it has long been observed that there is no single algorithm that performs best on all problems or even on all instances of the same problem [39, 29, 27]. To solve difficult computational problems effectively, it is often beneficial to utilise a *portfolio of algorithms* with complementary strengths. This gives rise to the field of Automated Algorithm Selection (AAS), where the aim is to automatically select the best algorithm(s) from an algorithm portfolio for a given problem instance. Over the last few decades, AAS has been shown to be very successful in various applications across a wide range of domains, including Boolean Satisfiability (SAT)[47], Constraint Programming (CP) [37, 33], AI planning [45], and combinatorial optimisation [30].

In the CP domain, an algorithm can be seen as a constraint solver (or a specific parameter configuration of a solver). Several studies have demonstrated complementary strengths of constraint solvers [17, 16] and the advantage of using them in combination in a portfolio setting [37, 8, 9]. However, the concept of a CP algorithm can be extended beyond the scope of a constraint solver, which often works on a low-level representation of a problem.

Those representations are usually less user-friendly and require specific modelling choices to be made about various parts of the problem description. To aid the modelling phase of combinatorial problems, mid-level and high-level constraint modelling languages such as MiniZinc [35] and ESSENCE [21] have been proposed. Accompanying these languages are modelling toolchains that support the automated translation of a mid-level or high-level representation of a problem to the low-level input supported by constraint solvers, such as the MiniZinc Toolchain [35], CONJURE [2], and SAVILE ROW [36]. The translation process involves several modelling and reformulation choices. Making the right combination of modelling and reformulation choices may have a significant impact on the performance of the target constraint solver [2]. In this context, we can consider an algorithm as a combination of modelling and reformulation configuration and a specific constraint solver.

Compared with the traditional viewpoint of seeing an algorithm as just a constraint solver, the extended viewpoint as a combination of modelling and solver choices can result in substantial improvement in the performance of AAS approaches. However, challenges arise when adopting AAS techniques for this extended context. More concretely, AAS techniques often rely on training Machine Learning (ML) models to predict the best algorithm(s) for a given problem instance based on the instance features. As in any ML application, having a good set of input features is of critical importance. The extracted features must be informative and relevant to not only the given problem instance but also to the performance landscape of the combination of modelling and solver choices on that instance.

One of the well-known instance features for constraint models are the fzn2feat features [6]. This is a set of 95 features that can be extracted from a representation of a constraint model written in the FlatZinc modelling language [35]. However, FlatZinc models are low-level representations and can only be obtained after specific decisions on the modelling and reformulation choices have been made. The features extracted are therefore tied to a specific low-level model, which may not be suitable for the task that we aim for, i.e., learning to select among different combinations of low-level models and solvers.

In this work, we propose to extract features from the high-level representation of a constraint problem. Instead of having to translate a given problem instance into a low-level representation (i.e., FlatZinc representation) before extracting (fzn2feat) instance features, our approach leverages language models to automatically learn instance features directly from the high-level representation of the problem instance. Compared with the existing fzn2feat feature extraction approach, our approach offers three advantages. First, in contrast to fzn2feat where the features were hand-crafted, our approach learns instance features automatically from the textual description of a problem instance. Second, fzn2feat relies on a specific low-level representation of a problem in FlatZinc, while our approach works directly at a high-level representation, which can potentially offer more information for the task of choosing the best combination of models and solvers. Third, as shown empirically, the proposed features, once learnt, are computationally cheaper to extract compared to fzn2feat features. We demonstrate our approach using the ESSENCE constraint modelling language via a case study with the car sequencing problem [22].[1]

In the rest of the paper, after giving the necessary background and discussing the related work in Section 2, we introduce in Section 3 our approach to AAS and in Section 4 our case study. Then we present in Section 5 the experimental evaluation of our approach and finally conclude in Section 6.

[1] `https://www.csplib.org/Problems/prob001/`

2 Background and Related Work

Constraint Modelling Tools To facilitate the modelling phase of combinatorial problems in CP, several domain-specific languages have been developed. Notable among these are MiniZinc [35] and ESSENCE [21]. ESSENCE is a high-level language designed to abstract problem modelling using a blend of natural language and discrete mathematics. This abstraction addresses the challenging nature of problem modelling, which demands expertise and domain-specific knowledge. CONJURE [2], a tool designed for ESSENCE, incrementally refines an initial ESSENCE model into ESSENCE PRIME, a lower level solver-independent constraint modelling language [36], through a series of transformations. Non-trivial transformations may yield multiple effective refinements, resulting in a portfolio of models with varying performances depending on the specific instance and solver used. This creates a complex landscape for selecting the optimal algorithm (ESSENCE PRIME model and solver combination).

ML for Algorithm Configuration and Selection Algorithm configuration is a field focused on optimizing the hyperparameters of an algorithm to enhance its performance based on criteria such as speed, memory usage, or accuracy. This process is essentially a search problem within the hyperparameter space, evaluated against a set of training instances [38]. Complementary to this is the field of algorithm selection, which involves choosing the best-performing algorithm from a portfolio of pre-tuned options to solve a specific problem instance [32]. Both algorithm configuration and selection often leverage ML techniques to inform their decision-making processes.

ML algorithms like random forests [12] and support vector machines [43] are particularly effective at identifying patterns in input features to predict optimal output, making them well-suited for these tasks. An ML algorithm takes as input a set of data points represented by a set of input features and their corresponding desired output (dataset). The initial dataset is analyzed by the algorithm that produces an ML model designed to address the desired task with a certain degree of correctness in the output. Essentially, an ML model is a function approximation from the feature input space to the desired output space. The efficiency of ML models in algorithm selection has been demonstrated in numerous applications [32, 47].

Neural Networks and Language Models Neural Networks (NNs) represent a powerful paradigm within ML, renowned for their ability to learn complex patterns from large datasets. They are particularly adept at generating features from textual input data [20], which simplifies the creation of ML models. Since the introduction of AlexNet in 2012 [31], NNs have been successfully applied to a wide array of tasks, such as image classification [41], text classification [46], robotics [14], and environmental science [34].

Related Work. Many AAS tools have been proposed to tackle CSPs. Most notably, SUNNY [33] and CPHydra [13] use a k-NN approach to compute a schedule of solvers which maximizes the chances of solving an instance within a given timeout, while Proteus [23] is a hierarchical portfolio-based approach to CSP solving that does not rely purely on CP solvers: it may choose a SAT solver along with an accommodating CSP-to-SAT translation to solve an instance. Moreover, AAS tools designed for SAT problems can be easily adapted to tackle CSPs (and vice-versa). An empirical evaluation of different AAS approaches for solving CSPs (including SAT portfolios) can be found in [5] and [7], which show empirical comparisons between SUNNY and AAS approaches originally proposed for SAT scenarios, such as 3S [26] and SATzilla [47].

Language models have previously been applied in CP to generate models from natural language problem descriptions [44, 4]. NNs have been used to learn features from the raw trajectories of search algorithms for selecting heuristic algorithms in bin packing problems [3].

Most relevantly, they have been employed to learn instance features for specific problems, such as the Traveling Salesman Problem (TSP), using transformer architectures [40]. In contrast, our contribution is designed to extract instance features from any Essence problem specification.

3 Methodology

Recall that given a problem class instance written in Essence and a set of constraint solvers, we can generate a *portfolio of algorithms* for the instance, where each algorithm is a combination of an Essence Prime model and a solver. The aim of our AAS task is to build a prediction model to select from the portfolio the best algorithm (with shortest runtime) for the instance. This task involves two key steps: (i) learning features representing a given Essence instance from its raw text content ; and (ii) using the learnt features to predict the best algorithm.

To address the first step, we propose to employ a Neural Network (NN) that encapsulates a language model to deal with text input. This approach has many advantages. First, language models like Bert have been proven effective in capturing high-level language features [20], eliminating the need to run a solver to extract the necessary features. Second, NN models can automatically generate the necessary features by starting from the raw input. This eliminates the need for handcrafting an effective feature set.

For the second step, we consider different options. A possibility is to combine the two steps and address the entire AAS task using a single NN. In this case, the probability associated to an algorithm by the NN indicates its likelihood of being the best and thus the one with the highest probability is deemed as the best. Another possibility is to detach the second step from the first and adopt an ML-based algorithm selector. This gives flexibility in the algorithm selection method, allowing us to leverage state-of-the-art tools as well as to experiment with others. In this case, the probability associated with an algorithm indicates its likelihood to be *competitive* (that exhibits good performance on the given instance). The algorithms along with the produced features are then given as candidates to the algorithm selector which then decides the best one. Both approaches are depicted in Figure 1, the details of which are explained in the following subsections.

3.1 Feature Learning Using a Language Model

We adopt a language model, a particular NN architecture, to learn a set of features that will be later used to select an algorithm in both options mentioned previously. The input of such a model is the raw text of the Essence instance in tokenized form (where each input word and symbol are transformed into a number), and the output is a feature vector that describes the semantic meaning of the input. In particular, we use an 8-bit-quantized [48] version of Longformer[10].[2] This is a Bert-like [20] architecture whose main advantage is the larger input size (2048 tokens instead of 512) and it has been proven competitive for a fine-tuning task [28]. In addition to the language model, the NN encapsulates a linear layer to process the features and produce an output. The linear layer comprises a neuron for each of the possible algorithms to choose from. Each neuron receives as input the feature vector produced by the language model. Then it computes the dot product with the learnt weights and adds a bias. The final result is a floating point value for each neuron.

[2] https://huggingface.co/tororoin/longformer-8bitadam-2048-main

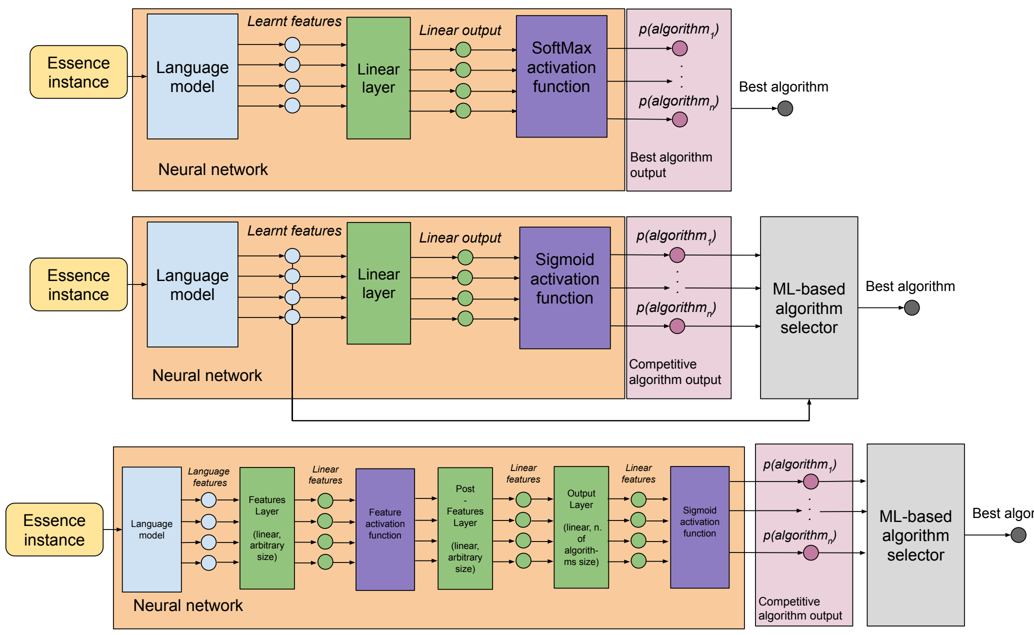

Figure 1 Two possible algorithm selection approaches: entirely NN-based (top) versus hybrid of an NN and an ML-based algorithm selector (middle). On the bottom the *new* version. The *Feature activation function* can be either Integer(ReLu), Integer(Sigmoid) or Integer(Tanh)

The main difference between the network of the first method (entirely NN-based) and the second one (hybrid of NN and ML-based algorithm selector) lies in the activation function that can transform the output of the linear layer from floating-point values into probabilities to better interpret the NN output. In the entirely NN-based approach, we want to learn a probabilistic distribution which has, as the most probable value, the best algorithm to choose. To achieve this output, we use the *softMax* activation function that transforms the input sequence into a probability distribution. In the hybrid case instead, we train the NN on a multi-label classification task, where the output comprises probabilities for each algorithm, indicating their *competitiveness* fraction. A higher probability suggests that the algorithm is less likely to be competitive. We consider an algorithm to be competitive if it solves an instance in less than ten seconds or in less than double the time taken by the best-performing algorithm for that instance. For example, if the best algorithm takes 15 seconds, any algorithm that completes the task in under 30 seconds is deemed competitive. To obtain such output, we use the *sigmoid* activation function which transforms each input value to a proper fraction, depending on its magnitude.

3.2 Algorithm Selection Using the Learnt Features

Once the NN is trained, the best algorithm for a given ESSENCE instance is chosen based on the probabilistic NN output. In the entirely NN-based approach, it is the one with the highest probability. In the hybrid approach, the probabilistic NN output is fed as input to an ML-based algorithm selector.

As an algorithm selector, we can rely on well-known methods such as Autofolio [32] and K-means clustering [1]. The first is a state-of-the-art tool that tunes the underlying model and its hyperparameters to optimize the performance. It can be used both for classification and regression tasks. The second is a clustering algorithm that assigns a cluster to a new instance. As features, these methods can exploit both the language model output and the probabilistic NN output. The features derived from the language model would be useful because they are trained on a similar task, capturing the general semantic structure of the instance. Whereas, the linear layer output indicates which algorithms are most likely to perform competitively. By combining the two, the features can encapsulate both a broad semantic representation of the instance and a specific prediction of the algorithms most likely to be competitive. To combine the features, the two outputs are concatenated, resulting in a vector of floating-point values for the given instance.

To obtain an algorithm selector from K-means clustering, each cluster is associated with the algorithm that resulted the best for the subset of the instances composing the cluster. At prediction time, a new instance is assigned to a cluster and the respective algorithm is selected.

As an alternative ML-based algorithm selector, we can use the probabilistic NN output as an initial filtering mechanism to eliminate the algorithms that are less competitive for a given instance, for instance those with probability less than 0.5. After the filtering, we can rank the remaining candidates based on a certain criterion (measured on the training set) and select the first ranked as the best algorithm. Possible criteria could be the overall performance or the number of instances where the algorithm wins.

4 A Case Study with the Car Sequencing Problem

We evaluate the performance of our approach to AAS using the Essence modelling language with a case study involving the car sequencing problem. In this section, we describe the case study. We start with the problem description in Essence and the instance set employed in the evaluation. We then present the combinations of (low-level) Essence Prime models produced by Conjure and constraint solvers, giving rise to a portfolio of algorithms to choose from. Finally, we describe how we obtain a dataset starting from the instance set and the algorithms, and discuss its suitability for an AAS task.

4.1 Problem Description and Instance Set

A series of cars are scheduled for production, each varying due to the availability of different optional features. The assembly line consists of various stations that install these options, such as air conditioning and sunroofs. Each station is designed to handle only a specific percentage of the cars passing through. To ensure that the workload at each station remains manageable, cars requiring the same option must be distributed evenly along the assembly line; clustering of these cars must be avoided to prevent overwhelming any single station. Therefore, cars must be sequenced so that the capacity of each station is not exceeded. For example, if a particular station can only manage a maximum of 50% of the cars passing through, the sequence must ensure that at most one car in every two requires that option. This sequencing problem is known to be NP-complete [22]. An Essence model for this problem is shown in Figure 2.

The Essence model defines three integer parameters *n_cars*, *n_classes*, and *n_options* representing the number of cars, classes of cars, and options available, respectively. Using these, three integer domains are defined: *Slots*, *Class*, and *Option*. These domains are used when defining further parameters and decision variables in the model as well as in constraint expressions. Three parameters with function domains are defined to represent the *quantity*

```
given n_cars, n_classes, n_options : int(1..)
letting Slots  be domain int(1..n_cars),
        Class  be domain int(1..n_classes),
        Option be domain int(1..n_options)
given quantity : function (total) Class  --> int(1..),
      maxcars  : function (total) Option --> int(1..),
      blksize  : function (total) Option --> int(1..),
      usage    : relation of ( Class * Option )
find car : function (total) Slots --> Class
such that forAll c : Class . |preImage(car,c)| = quantity(c)
such that forAll opt : Option .
            forAll s : int(1..n_cars+1-blksize(opt)) .
              (sum i : int(s..s+blksize(opt)-1) .
                toInt(usage(car(i),opt))) <= maxcars(opt)
```

Figure 2 Essence model of the car sequencing problem.

of each class of car required, a maximum number of cars (*maxcars*) that can appear in any block of cars, and block size (*blksize*) for each option. The *usage* parameter is a relation that indicates which classes use which options.

The only decision variable (*car*) in the model is a mapping from car production slots to classes. The problem constraints are captured in two top-level constraints (denoted by the keywords *such that*). The first set of constraints ensures that the number of cars in each class matches the required quantity. The second set of constraints ensures that for each option, in any block of *blksize(opt)* consecutive cars, the number of cars requiring that option does not exceed *maxcars(opt)*.

For all experiments in this work, we make use of a large instance set from a previous work [42]. It is composed of 10,214 instances, generated using an automated instance generation tool AutoIG [18] for constraint problems, and is publicly available in the ESSENCE Catalogue [19].

4.2 Combinations of Models and Solvers

Our algorithm portfolio contains three alternative ESSENCE PRIME models and four state-of-the-art solvers. The solvers are Kissat, Chuffed, CPLEX, and OR-Tools CP-SAT, each chosen for their potential complementary characteristics in combinatorial optimization. Kissat [11] is a modern clause-learning Satisfiability (SAT) solver. Chuffed [15] is a Constraint Programming (CP) solver enhanced with clause learning. CPLEX [25] is a commercial Mixed-Integer Programming (MIP) solver that excels in solving problems that heavily use arithmetic constraints. OR-Tools CP-SAT [3] is a hybrid solver developed by Google that integrates clause learning, CP-style constraint propagation, and MIP solving methods.

We use SAVILE ROW [36] to target these solvers. SAVILE ROW is a modelling tool that converts problem models written in ESSENCE PRIME into the input format required by these solvers and optimises the models based on the characteristics of the specific instance being solved. The ESSENCE PRIME models are obtained using CONJURE [2] in its portfolio mode,

[3] https://developers.google.com/optimization/cp/cp_solver

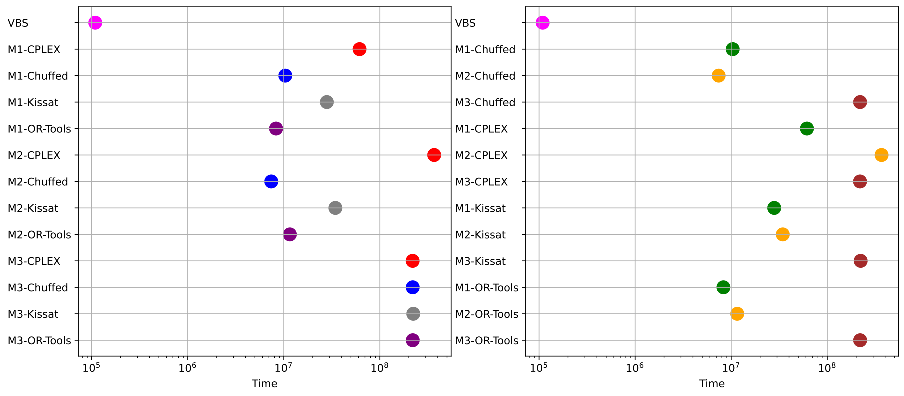

Figure 3 PAR10 value of each algorithm and the VBS on the instance set (lower is better), where the algorithms are grouped by their models (left) or solvers (right).

with variations arising from different representations for the *car* decision variable and the *usage* parameter, as well as the way problem constraints are formulated.

The *car* decision variable has two possible representations. The first is a one-dimensional array indexed by cars, containing decision variables with integer domains, where each entry represents the class selected for that car. The other is a two-dimensional Boolean array, indexed by both cars and classes, where a true value indicates the assignment of a car to a class. The *usage* parameter also has two possible representations: a two-dimensional Boolean array or a set of tuples. The second problem constraint in the ESSENCE model that refers to the *usage* parameter is refined with an *element* constraint when the Boolean array is chosen, instead with a *table* constraint when the set of tuples is chosen.

Using a combination of these model fragments, CONJURE constructs three distinct ESSENCE PRIME models. The first model M_1 has a one-dimensional array of integer variables for *car* and a set of tuples with a *table* constraint for the *usage* parameter. The second model M_2 couples the same one-dimensional array for *car* with a Boolean array for *usage* and the *element* constraint. The third model M_3 uses a two-dimensional Boolean array for *car*, and a set of tuples and the *table* constraint for *usage*.

4.3 Dataset and Algorithm Complementarity

The combination of three ESSENCE PRIME models and four constraint solvers results in a total of 12 algorithms. To perform the AAS task, we create a dataset by running the algorithms on the 10,214 car sequencing instances and record their runtime. The runtimes are measured on a computer with an AMD EPYC 7763 CPU, where each algorithm is given one CPU core and one hour of cut-off time per instance. We define the overall performance of an algorithm on a given instance set as the average runtime required to solve all the instances. To account for cases where an algorithm does not produce an answer within the given cut-off time, we adopt the Penalised Average Runtime (PAR10) metric from the AAS literature [32], where unsolved instances are penalised as 10 times the cut-off time. AAS techniques aim at *minimising* the PAR10 score.

To establish the potential of AAS in this case study, we analyze the performance of each algorithm on the instance set. Figure 3 shows the PAR10 score of the algorithms as well as the Virtual Best Algorithm (VBS), defined as the (hypothetical) algorithm selector that

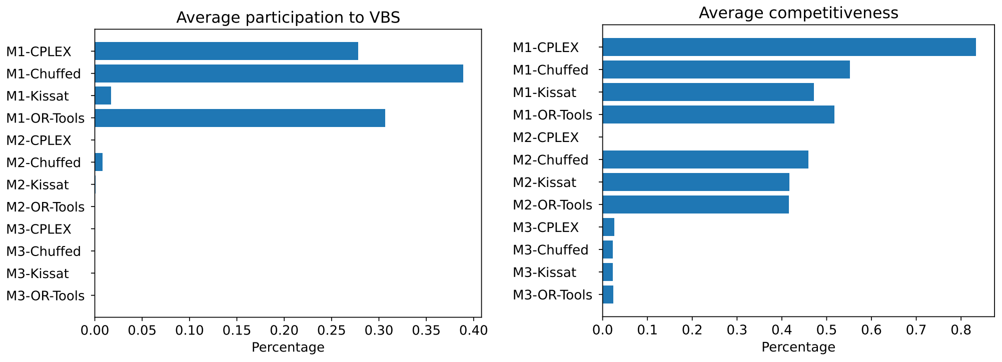

Figure 4 Average participation to VBS (left) and average competitiveness (right).

always correctly chooses the best algorithm for each instance. We see that, there is no model (resp. solver) that alone is always the best or worst independently of the coupled solver (resp. model). While M_2 is fastest with Chuffed, for M_1 it is OR-Tools, and these combinations are the two best algorithms. Even though M_3 has a much worse score with all the solvers, it does not take part of the worst algorithm, which is M_2-CPLEX. Except for the four algorithms involving M_3, they all exhibit different performances. Another observation is the big gap between the VBS and the best overall algorithm (M_2-Chuffed). We can therefore conclude that the algorithms have complementary strengths and leveraging them via AAS has high potential in this case study.

The complementarity of the algorithms in the portfolio can be further observed in Figure 4, where we plot on the left the average participation to VBS (as the percentage of the instances where the algorithm is the best) and on the right the average competitiveness (as the percentage of the instances where the algorithm is competitive). We can see that even though M_2-Chuffed appears as the best overall algorithm in Figure 3, it is the winner on a fairly small number of instances according to the left plot of Figure 4. Instead, M_1-CPLEX, M_1-Chuffed and M_1-OR-Tools have significantly higher numbers of instances where they win. These three algorithms cover a significant part of the instance space.

While many algorithms do not appear to participate at all to VBS, they are all competitive on some instances (with varying percentages), as shown in the right plot of Figure 4. An exception is M_2-CPLEX which in fact resulted as the worst overall algorithm in Figure 3. It is typically very difficult for an AAS method to always select the best algorithm for a given instance. At the same, this may not always be necessary, as competitive algorithms could also do well on the instance. We, therefore, expect that being able to choose a competitive algorithm for an instance increases the potential of AAS in our case study. Indeed, we will provide experimental evidence in Section 5 that AAS based on predicting the likelihood of an algorithm to be the best performs worse than predicting the likelihood to be competitive.

5 Experimental Evaluation

Having established the potential gain of AAS in the car sequencing case study, in this section, we experimentally evaluate the effectiveness of our approach.

The research questions (RQs) that we aim to answer in the evaluation are:

- RQ1: Can we learn an effective AAS model when combining feature learning and algorithm selection in a single NN model, or do we need to split the learning into two phases (as depicted in Figure 1)?

- RQ2: How do the learnt features perform on the AAS task compared to the existing fzn2feat features?
- RQ3: What is the feature extraction cost of the learnt features compared to the existing fzn2feat features?

We first describe in Section 5.1 how we trained the NN models and then present our study on each RQ in the subsequent sections.

The experiments are conducted using Python 3.11 in conjunction with PyTorch[4] and scikit-Learn[5] for the NN and the K-means clustering, while Python 3.6 was used with Autofolio. [6] The code is publicly available via the project repository. [7]

5.1 Neural Network Training

All NN models are trained on a GPU with Nvidia A5000 accelerator. [8] We trained each NN using a 10-fold cross-validation technique. At each fold, 10% of the dataset was used as a test set while the remaining 90% was split into training (90%) and validation (10%). For the approaches where the feature learning and algorithm selection are conducted separately, the same data split is used for the ML-based algorithm selector, therefore, if an instance was in the test set of the NN, it was also in the test set of the ML model that used the extracted features. Each network is trained for 10 epochs. For each fold, it took 57,328 seconds, which is around 15.9 hours, to complete the training of each network.

For the entirely NN-based approach where feature learning and algorithm selection are in a single NN model, the training is done using the typical cross entropy loss function for multi-class classification tasks. For the hybrid approach where the NN output is based on algorithm competitiveness, for the first 3 epochs, we used a learning rate of $1e^{-4}$ and, as a loss function, a weighted version of the Binary Cross-Entropy (BCE) loss that prioritised recall over precision. The formula of the weighted BCE loss function on each sample is shown in Equation (1), where n is the number of algorithms and y_i and $\hat{y}_i$ are the true and the predicted binary labels, indicating whether algorithm i is competitive or not. The first term in this formula represents the recall metric and is weighted twice over the second term.

$$\mathcal{L}_{\text{BCE}}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} [2y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{1}$$

For the next 6 epochs, we dropped the custom weights to use the normal BCE loss. The only notable change between epochs 3 to 6 and 6 to 10 was the change of learning rate that was $1e^{-4}$ for epochs 3 to 6 and $1e^{-5}$ for the final 4 epochs. For the whole training process, we used stochastic gradient descent as an optimizer for the model.

We leave as future work a more systematic study of which training schedules and hyper-parameter configurations are best suited to our task. The current decision is based on a small manual tuning study. The intuition behind splitting the training into different phases is as follows. At the first stage of the training process (the first 6 epochs), we prioritise recall over precision. If an algorithm is not competitive but is predicted as so, it may be incorrectly

[4] https://pytorch.org/
[5] https://scikit-learn.org/stable/index.html
[6] https://github.com/automl/AutoFolio/tree/master
[7] https://github.com/SeppiaBrilla/EFE_project
[8] https://www.nvidia.com/en-us/design-visualization/rtx-a5000/

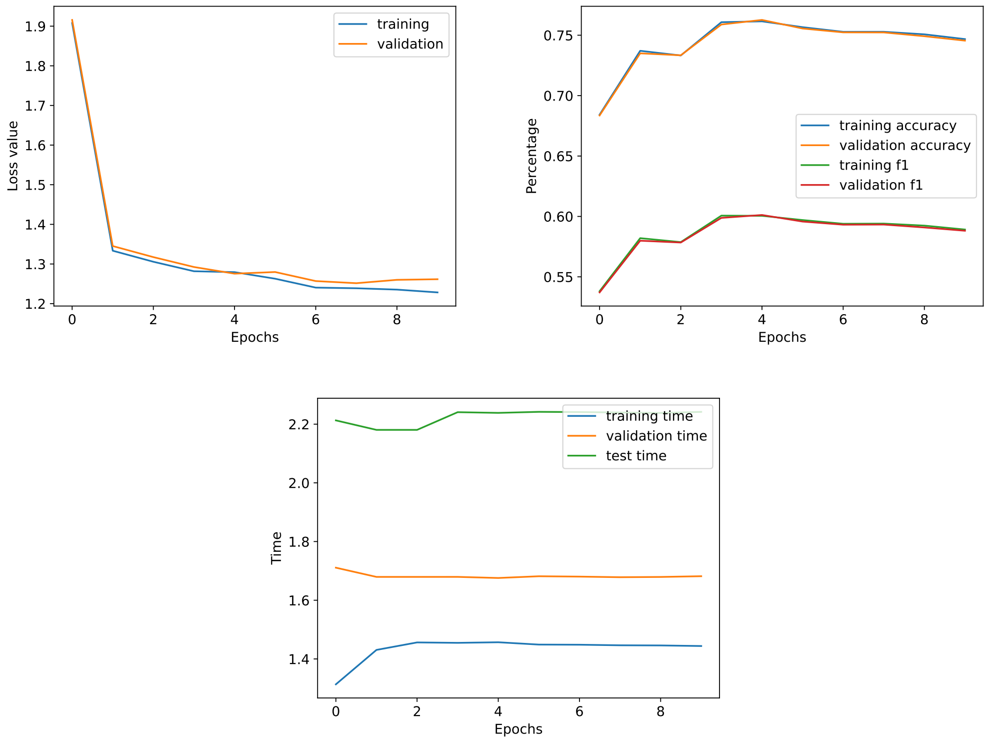

Figure 5 Training progress of the combined learning approach in one fold, shown by the cross entropy loss (top left), accuracy and F1 score (top right), and PAR10 score (bottom). The PAR10 score is normalised into the range $[0, 1]$ using M_2-Chuffed (the best overall algorithm) and the VBS.

chosen by the algorithm selector and could potentially result in a larger performance loss (in PAR10 score), therefore, the first term in Equation (1) is weighted higher to emphasise it.

5.2 Feature Learning and Algorithm Selection: Combining vs Splitting

In this section, we investigate RQ1: Can we learn an effective AAS model when combining both feature learning and algorithm selection in a single NN model, or do we need to split the learning into two phases (as depicted in Figure 1)?

Figure 5 presents an example of the training progress of the combined learning approach in one fold. Although the cross entropy loss value seems to indicate favourable results, the performance of the learnt network at each epoch in terms of accuracy and F1 score, as well as (normalised) PAR10 score, do not improve after the third epoch. The associated values in both training and the validation sets reach stagnation after that point. We observed the same pattern after having repeated the experiment across multiple folds. This observation highlights the challenges of training a combined learning approach for the AAS task.

One possible explanation for the difficulty of training is the fact that when treating the AAS task as a multi-class classification task, the training data is potentially highly imbalanced. For instance, some algorithms may win only on a small number of instances, making it difficult to predict them correctly, even though they may have a significant contribution to the overall PAR10 score of the algorithm selector. We mitigate this issue in our split learning approach by replacing the multi-class classification task with a multi-label classification task.

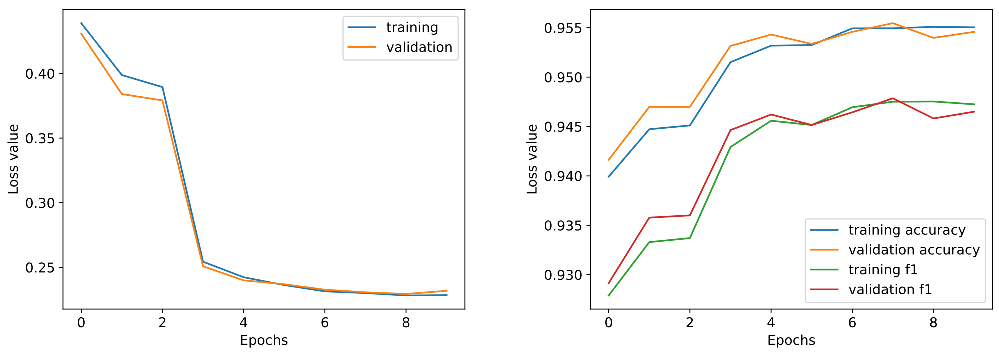

Figure 6 Training progress of the split learning approach in one fold, shown by the cross entropy loss (left), and accuracy and F1 score (right).

Instead of predicting the best algorithm, the output layer of our feature learning network will predict the competitiveness of each algorithm. In fact, this change allows us to train the network more effectively. As illustrated in Figure 6, the accuracy and F1 score now improve steadily during the training process. This study indicates that splitting the learning into two separate parts (feature learning and algorithm selection) is more effective. Therefore, we will adopt this approach in the remaining evaluation.

5.3 Learnt Features vs fzn2feat

In this section, we investigate RQ2: How do the learnt features perform on the AAS task compared to the existing fzn2feat features?

Our learnt features are the concatenation of the language model output and the probabilistic output of the NN, as illustrated in the bottom part of Figure 1. As an ML-based algorithm selector, we adopt AutoFolio [32] and K-means clustering [1], as mentioned in Section 3.2. With these selectors, we can use either our NN-based or the fzn2feat features. We refer to the four possible combinations as NN-Autofolio, fzn2feat-Autofolio, NN-Kmeans, and fzn2feat-Kmeans.

In addition to the algorithm selectors named above, our feature learning method offers other possibilities for algorithm selection. As a by-product of the feature learning process, we have a prediction model that tells us which algorithms are less competitive (with probability less than 0.5) for a given instance. As described in Section 3.2, this information can be used to filter out the less-promising algorithms for that particular instance. Among the remaining ones, we can select the best algorithm based on a specific criterion (measured on the training set), such as the PAR10 score or the number of instances where the algorithm wins. We refer to these simple selection approaches as NN-based Single Best Selection (NN-SBS) and Winner Selection (NN-WS), respectively.

Figure 7 presents the PAR10 scores of all the approaches described above on the training, validation and test sets across 10 folds. All four approaches using algorithm selectors surpass the performance of M_2-Chuffed (the best overall algorithm) and the two other simple selection methods (NN-SBS and NN-WS), confirming the effectiveness of learning AAS models using either feature set. Interestingly, AutoFolio offers significantly better performance than K-means on the training and the validation sets, but its generalisation is reduced as K-means is able to close the gap on the test set.

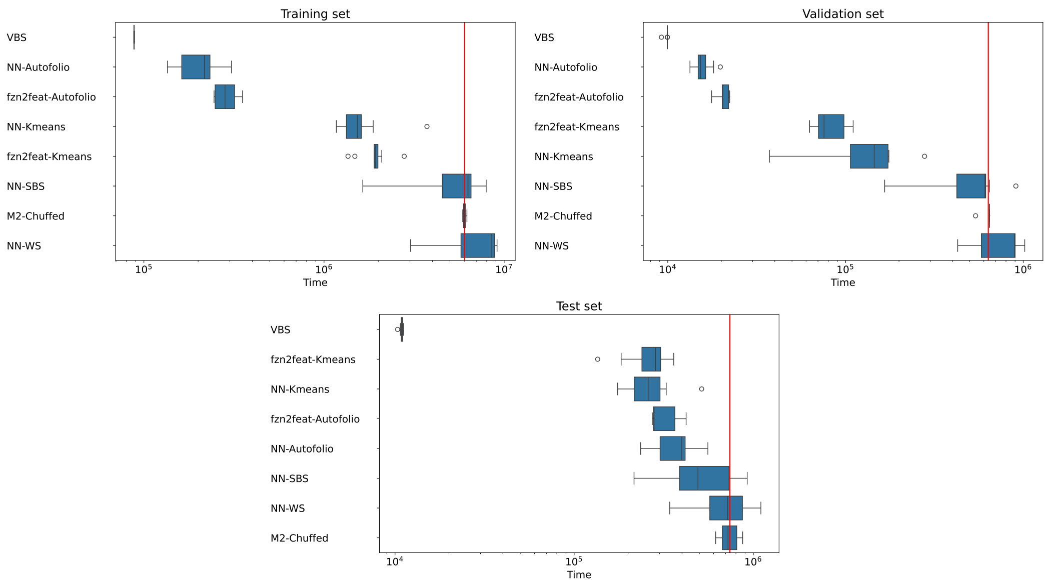

Figure 7 PAR10 scores of different AAS approaches across 10 folds. M_2-Chuffed is the best overall algorithm in the portfolio and its mean PAR10 score is shown with the red line. Reported prediction time includes the feature computation time.

Compared to fznfeat, our learnt feature set provides competitive performance, which indicates the effectiveness of the NN-based feature learning process. When combined with K-means, our feature set provides better overall performance on all the training, validation and test sets, although the difference between the two becomes less visible on the test set. When combined with AutoFolio, the fzn2feat methods offer slightly better average performance on the test set, although the learnt features do produce better on some folds.

AutoFolio is an algorithm selector that incorporates multiple state-of-the-art candidate ML models. It comes with a default selection model and that is what we have adopted in all the experiments so far. This is not necessarily the best choice, as the best model can depend on the specific scenario. AutoFolio includes an option to search in the vast space of several ML models and for their hyper-parameter configuration using the hyper-parameter optimisation tool SMAC [24]. To investigate the effectiveness of the two feature sets further, we conducted a new set of experiments where we allowed AutoFolio to be tuned. The tuning is done using SMAC in a 10-fold cross validation fashion. We let SMAC run for a maximum amount of 5 CPU hours on a machine with an AMD EPYC 7763 CPU.

Figure 8 shows the PAR10 scores of AutoFolio coupled with either feature set, with and without tuning. The tuning is very effective when the fzn2feat features are used as input. Surprisingly, when the NN-based features are used, there is a large variance in the performance of the tuned version on all three datasets. One potential explanation for this observation is that the number of features obtained from NN is very high (783 features) compared to fzn2feat (only 95 features). AutoFolio makes use of classical ML models such as random forests, and those might not be best suited to work on a very high dimensional input space. There are two potential ways to mitigate this issue. First, instead of using AutoFolio, we can try developing an NN-based algorithm selector, which may be better suited to be used with our learnt features. Second, we can try reducing the amount of features produced by the language model by imposing additional linear layers between the language model and the output layer, which may help to compress the learnt feature space. We leave the

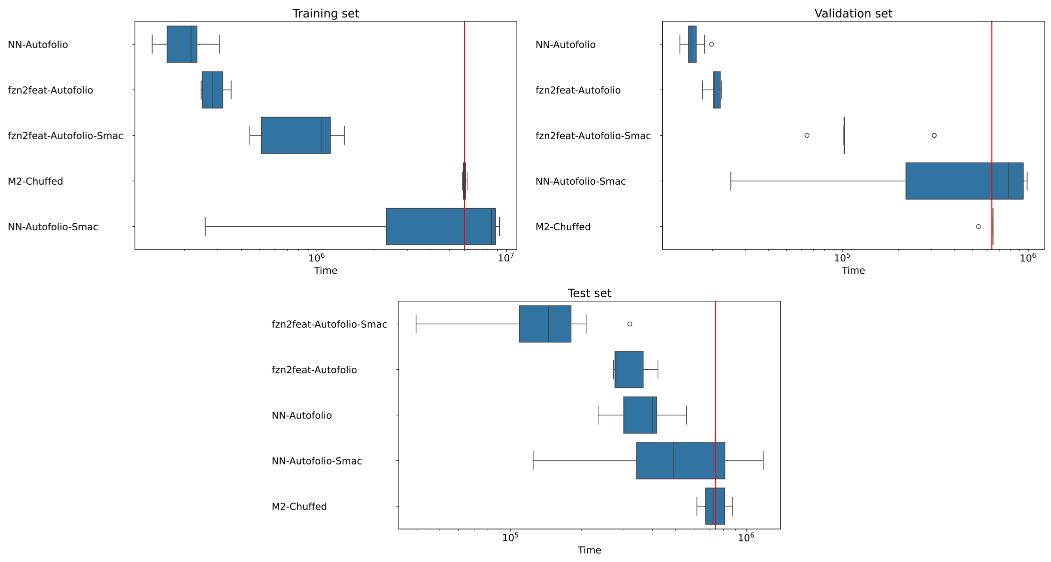

Figure 8 PAR10 scores of Autofolio (tuned with SMAC or not) across 10 folds. M_2-Chuffed is the best overall algorithm in the portfolio and its mean PAR10 score is shown with the red line. Reported prediction time includes the feature computation time.

	Median	Mean	Max	Min
fzn2feat	6.71	5.38	33.68	0.80
NN	0.02	0.02	0.38	0.02

Table 1 Statistics to compute a feature vector in seconds across all the instances.

investigation of these options for future work.

5.4 Feature Extraction Cost

In this section, we investigate RQ3: What is the feature extraction cost of the learnt features compared to the existing fzn2feat features?

As indicated in Table 1, a significant advantage of the NN-based approach is the time required to extract features from an instance. It consistently took less than 0.38 seconds to produce a result, whereas fzn2feat took up to 33 seconds. However, it is important to note that this speed advantage is contingent on the availability of a discrete graphics card, as NNs perform faster on GPUs.

6 Conclusions

In this paper, we explored the use of automatic feature learning for algorithm selection in the context of the car sequencing problem, leveraging the high-level constraint modelling language Essence. Our approach employed a language model to learn instance features directly from the problem descriptions, which were then used to predict the best algorithm for solving each instance.

Our experiments demonstrated that the learnt features could effectively be utilized within two different algorithm selection strategies (AutoFolio and K-means clustering). Both strategies showed promise, but each had its own strengths and weaknesses. The tuning experiments with AutoFolio highlighted the importance of careful feature set selection and tuning, especially given the high dimensionality of the learned features.

Despite these challenges, our results indicate that NN-based feature extraction offers a viable and efficient alternative to traditional methods, with significantly lower computational costs for feature extraction. However, the instability observed in the performance of tuned AutoFolio with NN-based features suggests further refinements are necessary. Future work could involve developing an NN-based algorithm selection approach tailored to handle high-dimensional feature spaces more effectively or incorporating feature compression techniques to enhance stability.

Overall, this study highlights the potential of ML and automatic feature learning in enhancing algorithm selection processes for combinatorial problems, paving the way for more adaptive and efficient solving techniques in various application domains.

Acknowledgements

This work was supported by the European Union's Justice programme, under GA No 101087342, POLINE (Principles Of Law In National and European VAT) and by a scholarship from the Department of Computer Science and Engineering of the University of Bologna.

References

1 Mohiuddin Ahmed, Raihan Seraj, and Syed Mohammed Shamsul Islam. The k-means algorithm: A comprehensive survey and performance evaluation. *Electronics*, 9(8):1295, 2020.

2 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, 2022.

3 Mohamad Alissa, Kevin Sim, and Emma Hart. Automated algorithm selection: from feature-based to feature-free approaches. *Journal of Heuristics*, 29(1):1–38, 2023.

4 Boris Almonacid. Towards an automatic optimisation model generator assisted with generative pre-trained transformer. *arXiv preprint arXiv:2305.05811*, 2023.

5 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, pages 316–324. Springer, 2013.

6 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*, pages 1357–1359, 2014.

7 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

8 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*, pages 1861–1867, 2015.

9 Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *Annals of Mathematics and Artificial Intelligence*, 76:229–246, 2016.

10 Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

11 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

12 Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

13 Derek Bridge, Eoin O'Mahony, and Barry O'Sullivan. Case-based reasoning for autonomous constraint solving. In *Autonomous search*, pages 73–95. Springer, 2012.

14 Matthew Browne and Saeed Shiry Ghidary. Convolutional neural networks for image processing: an application in robot vision. In *Australasian Joint Conference on Artificial Intelligence*, pages 641–652. Springer, 2003.

15 Chuffed Developers. Chuffed, a lazy clause generation solver. `https://github.com/chuffed/chuffed`. Accessed: 2024-07-05.

16 Nguyen Dang. A portfolio-based analysis method for competition results. *arXiv preprint arXiv:2205.15414*, 2022.

17 Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, and Peter Nightingale. A Framework for Generating Informative Benchmark Instances. 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 18:1–18:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2022.18`, `doi:10.4230/LIPIcs.CP.2022.18`.

18 Nguyen Dang, Özgür Akgün, Joan Espasa, Ian Miguel, and Peter Nightingale. A framework for generating informative benchmark instances. *arXiv preprint arXiv:2205.14753*, 2022.

19 Conjure developers. Essencecatalog: A collection of problem specifications in essence, 2024. Accessed: 2024-06-30. URL: `https://github.com/conjure-cp/EssenceCatalog`.

20 Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

21 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, 2008. `doi:10.1007/s10601-008-9047-y`.

22 Ian P Gent. Two results on car-sequencing problems. *Report University of Strathclyde, APES-02-98*, 7, 1998.

23 Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A hierarchical portfolio of solvers and transformations. In *Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings 11*, pages 301–317. Springer, 2014.

24 Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pages 507–523. Springer, 2011.

25 IBM. Ibm ilog cplex optimization studio: Cplex optimizer. `https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer`. Accessed: 2024-07-05.

26 Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17*, pages 454–469. Springer, 2011.

27 Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

28 Anant Khandelwal. Fine-tune longformer for jointly predicting rumor stance and veracity. In *Proceedings of the 3rd ACM India Joint International Conference on Data Science & Management of Data (8th ACM IKDD CODS & 26th COMAD)*, pages 10–19, 2021.

29 Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pages 149–190, 2016.

30 Lars Kotthoff, Pascal Kerschke, Holger Hoos, and Heike Trautmann. Improving the state of the art in inexact tsp solving using per-instance algorithm selection. In *Learning and Intelligent Optimization: 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers 9*, pages 202–217. Springer, 2015.

31 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

32 Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

33 Tong Liu, Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. sunny-as2: Enhancing sunny for algorithm selection. *Journal of Artificial Intelligence Research*, 72:329–376, 2021.

34 Holger R Maier and Grame C Dandy. Neural network based modelling of environmental variables: a systematic approach. *Mathematical and Computer Modelling*, 33(6-7):669–682, 2001.

35 Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

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

37 Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish conference on artificial intelligence and cognitive science*, pages 210–216, 2008.

38 Rong Qu. A general model for automated algorithm design. *Automated Design of Machine Learning and Search Algorithms*, pages 29–43, 2021.

39 John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

40 Moritz Vinzent Seiler, Jeroen Rook, Jonathan Heins, Oliver Ludger Preuß, Jakob Bossek, and Heike Trautmann. Using reinforcement learning for per-instance algorithm configuration on the tsp. In *2023 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 361–368. IEEE, 2023.

41 Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

42 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023.

43 Shan Suthaharan and Shan Suthaharan. Support vector machine. *Machine learning models and algorithms for big data classification: thinking with examples for effective learning*, pages 207–235, 2016.

44 Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy grail 2.0: From natural language to constraint models. *arXiv preprint arXiv:2308.01589*, 2023.

45 Mauro Vallati, Lukáš Chrpa, and Diane Kitchin. Asap: an automatic algorithm selection approach for planning. *International Journal on Artificial Intelligence Tools*, 23(06):1460032, 2014.

46 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

47 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.

48 Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. Quantization networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7308–7316, 2019.