Optimal decision trees for the Algorithm Selection Problem

A dynamic programming approach

Giulio Segalini[1]

Supervisor(s): Emir Demirović[1]**, Jacobus G.M. van der Linden**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 20, 2023

Name of the student: Giulio Segalini Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Jacobus G.M. van der Linden, Burcu Özkan

Abstract

The Algorithm Selection Problem is a relevant question in computer science that would enable us to predict which algorithm would perform better on a given instance of a problem. Different solutions have been proposed, either using Mixed Integer Programming or machine learning models, but both suffer from either poor scalability, no guarantees of optimality, or not interpretable models that could be used to gain insights into the nature of the problem.

In this work we propose a dynamic programming method to build Optimal Decision Trees to solve the Algorithm Selection Problem, giving us an interpretable model that is globally optimal over the training dataset. We also show that this method is orders of magnitude faster in training trees that are identical to the current state-of-the-art and propose possible improvements for future work.

1 Introduction

Computationally-hard problems are critical in computer science, and being able to solve them optimally has been a major area of interest for decades. A particular subset of appeal is NP-Hard problems, the set of all problems at least as hard as the hardest problem in NP, for which no determinist polynomial time algorithm is known. This implies that many possible algorithms have been developed over the years, but given that none of them has strictly better asymptotic runtime than the others, it is important to analyze which one could be the best option in specific scenarios.

An example of one of these problems is SAT, the satisfiability problem, the problem of finding an assignment of boolean variables that satisfies a set of propositions. This problem was proven to be NP-complete [6] and so being able to solve it efficiently is of utmost importance. For this reason, competitions are hosted each year to compare different algorithms. The results of these competitions show us that there is no best way to solve the problem and performance varies considerably in different instances [2].

The problem of assigning instances of a problem, for example, the aforementioned SAT, to the best algorithm to solve them is known as the Algorithm Selection Problem, as defined by Rice [22]. This is usually solved using machine learning techniques and given that in recent years the need for interpretable models has become more pressing [23] decision trees have been studied.

Decision tree learning is a widely used approach in machine learning, as they provide a concise and humanly interpretable model. Moreover, given the nature of the model itself, it is possible to split the training into subtrees that can be trained independently, opening the way to possible performance improvements. Heuristic methods can be used to obtain trees with a high level of accuracy, but none of these are guaranteed to be globally optimal and so it may not necessarily be the best representation of the data in terms of accuracy, size, or other constraints. This motivated the development of different strategies for obtaining optimal decision trees, that is the best possible tree according to a specific metric, for example, the number of misclassified elements.

Different work has been done to achieve this, for example, the work of Demirović et al. [7] and van der Linden et al. [24] that exploit the recursive nature of the tree to train it using dynamic programming, Narodytska et al. [20] which models the problem as an instance of the SAT problem and Boas et al. [27] which uses Mixed Integer programming and Variable Neighborhood Descent. The dynamic programming approach was able to achieve high scalability compared to other methods, but as of now has been tested on specific classification problems. It would be interesting to adapt its ideas to solve the Algorithm Selection problem.

Research questions This paper shows how to generate optimal decision trees for the Algorithm Selection problem using dynamic programming and how this method compares to the current state of the art in terms of scalability.

Contributions This new method outperforms the current state-of-the-art by several orders of magnitude while being able to obtain the same level of accuracy. It is also able to train trees up to depth 7, while the competing method reported optimal trees up to depth three.

2 Related Work

This section shows a literature review of previous similar work, in particular on the Algorithm Selection problem and optimal decision trees.

2.1 Algorithm Selection Problem

The first definition of the Algorithm Selection problem (ASP) was given in 1976 by Rice [22]. One of the first approaches to solve the ASP used regression to predict the performance of the problem for a specific algorithm and select the best one [29] then Lagoudakis et al. [15] successfully used reinforcement learning and opened the way for machine learning methods exploration.

A common assumption used by different methods is that problem instances can be grouped into clusters of similar characteristics and that a specific algorithm will perform similarly on instances in the same cluster. This approach enables the ASP to be reduced into a training phase in which we identify the clusters and an association phase in which different algorithms get assigned to clusters. Using this approach Malitsky et al. [17] devised a method, later refined by Kadioglu et al. [12] by combining the use of a scheduler to maximize the number of solved instances in a set time. Malitsky et al. have then improved over the clustering technique by using Cost-Sensitive Hierarchical Clustering [18] and Kadioglu et al. [13] also devised a strategy to solve the ASP by obtaining instance-specific algorithm configuration values together with algorithm selection from other works. In 2022 Karahalion et al. [14] devised a method to not only choose algorithms but also the optimal variable ordering to solve Constraint Programming models.

Finally, Boas et al. [27] showed how to build optimal decision trees to solve the ASP by using Mixed Integer programming to build optimal trees and a heuristic based on Variable

Neighborhood Descent to obtain suboptimal trees, but in substantially less time. This last work shows the potential of decision trees to obtain accurate results but has significant difficulties in scalability. In particular, only Optimal trees up to depth three were trained, while heuristic-based methods train trees that perform better than Random Forests with more interpretable results.

2.2 Optimal Decision Trees

Optimal decision trees have been studied for an extended period, and in 1976 Laurent et al. proved that the problem of constructing them is NP-Complete [16]. Bennett et al. have proposed a solution for constructing globally optimal decision trees by fixing the structure of the tree and then solving a system of linear inequalities using existing optimizers [3].

The idea of using constraints programming or Mixed Integer Programming (MIP) together with specific optimization has been proposed multiple times, in particular by Bertsimas et al. in 2007 [5], again Bertsimas et al. in 2017 [4] and Verwer et al. [25], that the refined the method in 2019 by using an efficient binary encoding of the training dataset [26].

More recent work was conducted by Hu et al. [10] introducing a method to build optimal sparse trees, Aglin et al. [1] that presented DL8.5, an algorithm that uses Branch-and-Bound search, caching, pruning, and heuristics to outperform other works by order of magnitudes. Demirović et al. presented MurTree that employs "many specialized techniques that exploit properties unique to classification trees" [7].

All of these methods use dynamic programming to split the problem into smaller subtrees, but sacrifice generalizability to gain better runtime performance. Of the works stated above MurTree can obtain optimal classification trees and has been further generalized to any optimization task in STreeD [24], which is the starting point for this work.

3 Preliminaries

This section provides insights and explanations into the methods and terminology used in the current literature. The first part formally defines the algorithm selection problem, while the second part describes the dynamic programming formulation for building optimal decision trees.

3.1 Formal Problem Description

The Algorithm Selection Problem (ASP) has been first formally defined by Rice [22] as the problem of identifying the best algorithm to solve a particular instance of a problem according to a given metric, for example, runtime or memory usage. We can use this to delineate the following notation:

- $\mathscr{P}$: The problem space, that is the set $\{p_1, p_2, \ldots, p_n\}$ of all possible instances of the specific problem of which we are analyzing possible algorithms.
- p: A particular element of the problem space.
- $\mathscr{A}$: The algorithm space, that is the set $\{a_1, a_2, \ldots, a_m\}$ of all available algorithms that correctly solve instances in $\mathscr{P}$.
- a: An element of the algorithm space, one specific algorithm.
- $R(p,a)$: A mapping from a pairing algorithm and problem instance to the metric we want to optimize, in most cases this is runtime.
- $S(p)$: The solution to the problem, a mapping from a problem in the problem space to an algorithm in the algorithm space.
- $B(p)$: The ideal solution to the problem, the mapping that goes from each problem to their best performing algorithm according to the metric $\mathscr{R}$.

With the following notation, we can define the problem as finding the mapping $S(x)$ that minimizes the distance from the optimal mapping, giving us the following objective function:

$$\min \sum_{p \in \mathscr{P}} |R(p,S(p)) - R(p,B(p))|$$

This formalization is as general as possible and, in the following section, it is adapted for usage with the dynamic programming formulation of decision trees. However, it does not consider the concept of features, which is important to create meaningful mappings.

We can summarize the definition in a single sentence: given a set of algorithms and a new instance, which algorithm should we select based on its features such that the chosen algorithm optimizes a performance metric?

3.2 Optimal Decision tree for the ASP

This subsection covers the description of mappings and input and output variables, followed by an explanation of how features are defined and formalized.

Functions and Variables

The definition in subsection 3.1 can almost be employed as is, but we need to introduce a bit more notation to better adapt the problem for being solved with decision trees.

- $\mathscr{P}_{tr}$: A subset of the problem space used for training.
- $\mathscr{P}_{ts}$: A subset of the problem space used for testing.
- $R(p,a)$: Runtime of algorithm a on problem p relative to the lowest runtime, so $R(p,B(p)) = 0$.

The objective function is the same as before, but given that we construct Optimal Decision Trees, we will minimize its value over the training set, while the testing set is used to assess the accuracy of the obtained tree.

Features

To effectively distinguish problems, we identify features for each instance. We can define a feature as an intrinsic characteristic of the problem space that changes between instances of the same problem, for example, the number of propositions in a SAT instance.

Features vary widely between problem spaces, and we need good formalization. For them to be usable by the dynamic programming formulation of subsection 3.4, they must be binary. More details on how binarization is achieved are in section 5.1. The formulation below defines two feature spaces: one is the original, while the other is the binary version.

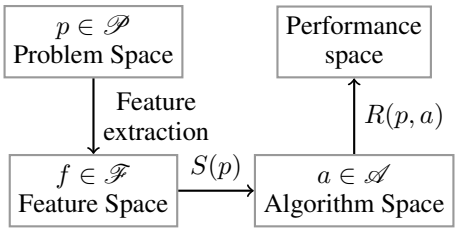


Figure 1: The Algorithm Selection problem [22; 27]

- $\mathscr{F}$: The feature space, the different characteristics we can quantify for each problem.
- f: A feature in the feature space. The feature can assume different values from $V_f = \{v_{f1}, v_{f2}, \ldots, v_{fk}\}$.
- $F(p) = (f_1, f_2, f_3, \ldots, f_k)$, the feature values for problem p.
- $\mathscr{F}_B$: The binarized feature space. Each feature in $\mathscr{F}$ gets binarized into n values, more details in section 5.1.
- f_b: A feature in the binarized feature space, this time it can only get values from $V_f = 0, 1$
- $FB(p) = (fb_1, fb_2, fb_3, \ldots, fb_{n \cdot k})$, the binary features value for problem p.

It is important to note that the cardinality of the feature space should be significantly less than that of the problem space, to mitigate the risk of overfitting by just looking at features that are unique to specific instances.

The formalization is summarized in Figure 1.

3.3 Decision Trees

A decision tree is a decision model that uses a tree structure in which a branching node represents a conditional statement, while a leaf node represents a decision. Given a new instance that needs to be analyzed using the model, we can traverse the tree and follow each branching point that satisfies the instance's features until we decide on a leaf node.

If we have a set of branching conditions F, for example, a set of binary features, and a set of labels K, we can see a decision tree as a tuple $\tau = (B, L, b, l)$, with B being the set of branching nodes, L the set of leaf nodes, $b : B \to F$ an assignment of a splitting condition to each branching node, and $l : L \to K$ the mapping of labels to each leaf node. Given any node $u \in B$ we can call u_l and u_r the left and right child respectively.

Using this definition, we can build a dynamic programming formulation of decision trees as shown in subsection 3.4.

3.4 Dynamic Programming Formulation

As said in Section 2, different approaches to building optimal decision trees have been studied, but in this research, we will work on the dynamic programming formulation from MurTree [7] and STreeD [24]. This has various advantages that are consequences of the ability of the problem to be broken into smaller sub-problems that can be cached and used again if needed. The sub-problems are also independently solvable and parallelizable. Furthermore, when a subtree is solved with a suboptimal score, we can prune the search space and avoid unnecessary computations.

To explain the formulation, we need to define a common terminology:

- $\mathscr{K}$: the set of possible labels.
- $opt(S)$: a function that given a set of solutions returns the set of optimal solutions, this requires a comparison $cmp(s_1, s_2)$ operator between solutions to identify which is best.
- $merge(S_1, S_2)$: a function that given two sets of solutions for left and right subtrees, returns the set of all solutions of trees that have one subtree from each set as children.
- $g(D,k)$: a function that given dataset D and a label k returns the total cost associated of assigning the label k to all instances in D.
- D_f and $D_{\bar{f}}$: respectively, the subset of D containing values that have positive value for feature f and values that have negative value for feature f.

The DP formulation is then shown in Equation 1 and uses the same ideas as STreeD [24].

$$T(D,d) = \begin{cases} opt\left(\bigcup_{k \in \mathscr{K}} g(D,k)\right) & d = 0 \\ opt(\bigcup_{f \in \mathscr{F}} \\ merge(T(D_f, d-1), T(D_{\bar{f}}, d-1))) & d > 0 \end{cases} \tag{1}$$

The equation consists of two cases. The base case occurs when $d = 0$, where we consider all possible labels and return the ones that yield optimal solutions. For all other depths, we return all optimal solutions built by splitting the dataset over a binary feature, getting the optimal sub-solutions, and merging them.

We can further optimize the dynamic programming approach by employing a specialized solver for trees of depth two or less, as first shown in MurTree [7] and STreeD [24]. This solver consists of two distinct phases: the frequency counting phase and the tree construction phase. The first phase counts the number of instances and their associated cost that present a specific set of features, while the second phase can use the just computed counts to find the best possible tree.

In more detail, during the frequency counting phase we can first compute the frequency count for every possible feature pair and label of the dataset and the associated cost of having that specific label assigned to the relevant instances. This will be called $C(f_1, f_2, l)$, the total cost of instances that present features f_1, f_2 when assigned with label l.

By using the sum of the costs computed before, we can find which triplet $(f_{root}, f_{left}, f_{right})$ of features and quadruplet of labels (l_1, l_2, l_3, l_4) gives us the best tree, as shown in Figure 2. The objective becomes then minimizing (or maximizing depending on the task):

$$\begin{aligned} &C(f_{root}, f_{left}, l_1) + C(f_{root}, \bar{f_{left}}, l_2) \\ &+ C(\bar{f_{root}}, f_{right}, l_3) + C(\bar{f_{root}}, \bar{f_{right}}, l_4) \end{aligned} \tag{2}$$

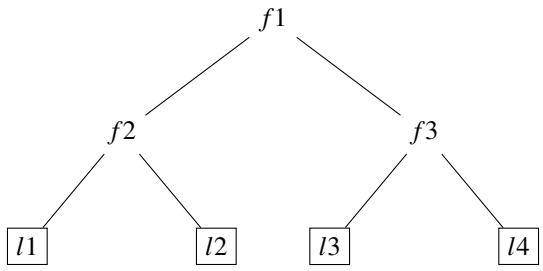


Figure 2: An example tree built using a quadruplet of labels and a triplet of features.

To prune the search space, we can keep track of upper and lower bounds solutions as shown in MurTree [7] using concepts from DL8.5 [1]. MurTree [7] also implements a stricter lower bound approach called similarity lower bound which STreeD [24] generalizes by using the largest change in score assigning a particular label to an instance would produce. These values are then employed to compute the maximum difference in score between two datasets.

The details of how this method works are not relevant to this paper. However, it is important to note that to use this technique, we only need to define the worst score associated with a specific label.

4 Contributions

In this section, we show the main contributions of this work. Firstly, we adapt the dynamic programming formulation from the previous section for the Algorithm Selection problem. Secondly, we describe the adaptation of the depth-two solver and the bounds from [7]. Lastly, we introduce a strategy to reduce overfitting by incorporating a regularization term.

The contributions are built upon the STreeD framework [24], and we introduce all the elements required to solve the Algorithm Selection problem efficiently and effectively.

4.1 Dynamic Programming for the ASP

As the definition of the dynamic programming formulation in subsection 3.4 is very general, it can be simplified and adapted for the Algorithm Selection problem by defining the cost function g as below:

$$g(D,k) = \sum_{p \in D} \mathscr{R}(p,k) \tag{3}$$

This function computes the sum of all differences between the best possible runtime and the runtime we would obtain with algorithm k for problems p in D.

We can then define a comparison operator $cmp(s_1,s_2)$ that returns whether solution s_1 is better than solution s_2 as:

$$cmp(s_1,s_2) = s_1 < s_2 \tag{4}$$

by which a solution is better than another if the total runtime is a smaller value.

4.2 Depth-two solver and Lower bounds

Using the comparison operator and the cost function, we can modify the specialized depth-two solver to meet the requirements of the new objective function. Instead of keeping track of frequencies that are used in the misclassification score, we can keep track of the cost $R(p,a)$ that assigning label (algorithm) a to problem p would give and then compute the value of $C(f_1,f_2,l)$ using Equation 5:

$$C(f_1,f_2,l) = \sum_{p \in D_{f_1,f_2}} R(p,l) \tag{5}$$

It is then possible to precompute these values for all combinations of features and labels and obtain the same speedups that MurTree [7] and STreeD [24] achieve.

In addition to the specialized solver for trees of depth two, MurTree [7] and STreeD [24] can use upper and lower bounding to prune the search space. In particular, STreeD can do so if it is possible to define the value w_k, the worst-case contribution assigning an instance to label k to the total cost. For the algorithm selection problem, this is:

$$w_k = \max\{\forall p \in D \mid R(p,k)\} \tag{6}$$

4.3 Reducing overfitting using a regularization term

To reduce the chance of overfitting we can introduce a regularization term [30] in which we penalize small leaf nodes in the same way Boas et al. [27] have proposed. This term introduces a penalty of β for each instance that is missing in a leaf node to reach a certain threshold τ, greatly reducing the chance of obtaining trees in which some algorithms are assigned to only a few instances. This changes our objective function to:

$$\min \sum_{p \in \mathscr{P}} R(p,S(p)) + \sum_{l \in \text{leaves}} d(l) \cdot \beta \tag{7}$$

In which $d(l)$ is the function that returns the distance between the number of instances assigned to leaf node l and the threshold τ, so:

$$d(l) = \begin{cases} 0 & \text{size of } l \geq \tau \\ \tau - (\text{size of } l) & \text{otherwise} \end{cases} \tag{8}$$

This added penalty changes the worst case added cost defined in Equation 6 as adding or removing an instance p to a leaf with label k can change the score by β. The new worst case needs to also account for this, becoming:

$$w_k = \max\{\forall p \in D \mid R(p,k)\} + \beta \tag{9}$$

5 Experimental Setup and Results

In this section first, we show which dataset and features are used, followed by the actual setup and each experiment's results. We conducted a total of 4 experiments to check respectively: scalability with increasing maximum allowed depth to compare against the work of Boas et al. [27], how much the bound-based pruning is effective, in and out-of-sample accuracy, and how different binarization strategies affect runtime.

5.1 Datasets and features

For this work, two datasets from MaxSAT competitions 2021[1] and 2022[2] were used, as they contain a realistic scenario in which competitive solvers are benchmarked against a wide variety of instances. The unweighted complete tracks from both years were used, as in this setting, the competing algorithms obtain the same result and the only performance metric is runtime. On the other hand, the weighted track measures both runtime and the total weight of satisfied clauses, which would require the score to combine these two values. To use the data with the dynamic programming method, we need to extract features and then binarize them.

Feature extraction

Feature extraction for SAT instances has been extensively studied, in particular, the work of Nudelman et al. [21] that has then been expanded by Hutter et al. [11] presented 138 possible features. We selected a subset of 32 features from the aforementioned work, prioritizing those that could be computed within a reasonable time frame. These highlighted features are shown in Figure 3.

These features can be grouped into three groups: features related to the size of the problem, features that show the instance balance through ratios of positive and negative literals and variable occurrences together with the number of short clauses, and features that relate to the Horn formula and clauses.

A clause is called a Horn clause when it's a disjunction of at most one positive literal [9]. These clauses are important in SAT instances as an instance containing only Horn clauses is solvable in linear time [8]. For most of these features the mean, variance, minimum and maximum values, and Shannon's entropy are included as they could provide more useful information.

Feature binarization

The dynamic programming formulation assumes binary features, while the feature extracted previously are continuous values. To obtain binary features without significant loss of accuracy, we can discretize the features into k bins and have each original value be encoded by $k-1$ new ones. First, all the possible feature values for the data are sorted and divided into k equally sized bins delimited by $k-1$ threshold values. Then we can use each new feature to represent whether the original feature value is smaller than the respective threshold value.

This approach is similar to what the work of Boas et al. [27] does, as we are effectively defining $k-1$ branching points for each feature. For the rest of this work, the number of bins was fixed to $k=2$, unless specified otherwise, to be able to train trees in a reasonable time. We explore different values of k in subsection 5.6 to showcase how they impact runtime and accuracy.

5.2 Experimental setup

The main reason to run experiments is to assess the scalability of this work against the current state-of-the-art optimal decision trees used to solve the Algorithm Selection problem, the method of Boas et al. [27]. As said in section 2, two methods are proposed in the work of Boas et al., but as this paper focuses on globally optimal trees only a comparison against the MIP model was executed.

- **Problem size features**
  1. Number of clauses (c)
  2. Number of variables (v)
  3-5. Ratios: c/v, c^2/v^2, c^3/v^3
  6-8. Reciprocals of above ratios: v/c, v^2/c^2, v^3/c^3
  9-13. Length of clauses: mean, variance, min, max, entropy
- **Balance features**
  14-18. Fraction of positive clauses: mean, variance, min, max, entropy
  19-21. Fraction of unary, binary, and ternary clauses
  22-26. Fraction of positive occurrences for each variable: mean, variance, min, max, entropy
- **Horn clauses features**
  27. Fraction of horn clauses
  28-32. Occurrences in horn clauses for each variable: mean, variance, min, max, entropy

Figure 3: Extracted features from SAT instances

The MIP model from [27] was implemented in Python 3 using Gurobi's[3] interface, a commercial MIP solver. The model defines features as continuous values that can be split on a specific set of numbers, This is effectively the same as what we presented in section 5.1, but done during training and not as a pre-processing step. To be able to use the same processed file format with features already divided into bins and represented by binary $0,1$ values, we define the branching values for each variable as the singleton set $\{0\}$.

The dynamic programming implementation is based on STreeD [24] and uses a C++ framework. As both methods use a regularization term based on parameters τ and β, values of $\tau = 10$ and $\beta = 50$ were chosen to compare fairly to the work in [27].

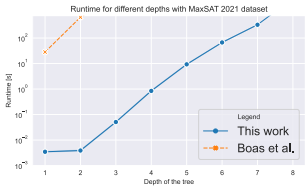
All experiments were run on a single thread on an Intel Core i7-10510U CPU with a timeout of 3600s. It is important to note that Gurobi and other MIP solvers support parallelization and that it can be used to obtain better performance, but for these experiments, it was chosen to have the same running condition as the dynamic programming implementation, which is not parallel.

The two datasets from subsection 5.1 were used to train trees of increasing depths and measure runtime, this enables us to understand how the different implementations scale on more complex trees. For both datasets, features were extracted and binarized, with instances in which none of the algorithms found a solution during the competition removed, giving us a total of 439 instances and eight labels for the 2021

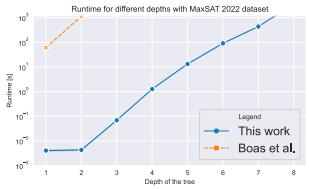
[1] https://maxsat-evaluations.github.io/2021/

[2] https://maxsat-evaluations.github.io/2022/

[3] https://www.gurobi.com/

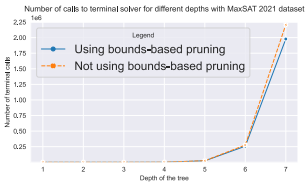


(a) Experiments with MaxSAT 2022 dataset

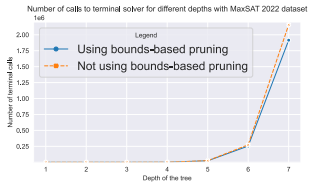


(b) Experiments with MaxSAT 2022 dataset

Figure 4: Runtime experiments run with two datasets

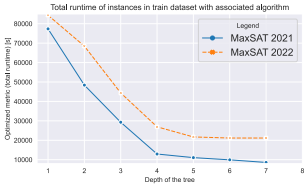


(a) Experiments for terminal calls run with the MaxSAT 2021 dataset

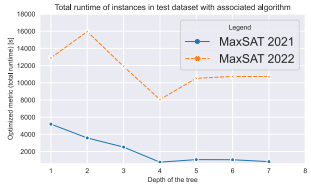


(b) Experiments for terminal calls run with the MaxSAT 2022 dataset

Figure 5: Experiments for terminal calls run with two datasets



(a) Metric value over training dataset for increasing depth



(b) Metric value over testing dataset for increasing depth

Figure 6: Experiments to check the total run time of instances and assigned algorithm for trees with increasing depth

dataset, while we have 461 instances and 11 labels for the 2022 one.

To test the efficacy of the bound-based pruning, we executed experiments in which calls to the depth-two solver were measured when the bound was calculated using Equation 9 and when no bounds were employed. Running this on both datasets with increasing maximum depths gives us an idea of the amount of pruning that can be done.

To showcase the usefulness of higher depth models two more experiments were run to show how the optimized metric value over the training and the testing set respectively changes for increasing depths. In each run, the datasets were split randomly into a training and a test set dedicating 90% the instances to the former and the rest to the latter.

The last set of experiments shows how different binarization strategies, in particular the number of bins, can influence the value of the metric we are trying to optimize.

5.3 Runtime performance analysis

The plots in Figure 4 representing the data collected from the experiments all use a logarithmic scale on the y-axis, as the runtime increases exponentially. We run both methods five times for each depth to show a 95% confidence interval and stopped as soon as one of the training timed out for the current depth.

Results for the 2021 dataset are shown in Figure 4a and results for the 2022 datasets are similar and shown in Figure 4b.

We can see how the dynamic programming implementation outperforms the MIP model, both in runtime for small depths and in the possibility of training deeper trees. In both experiments, the MIP implementation was unable to obtain a globally optimal tree of depth three in less than 1 hour, while the dynamic programming formulation needs less than a second for all depths smaller than 4.

The objective function value for the trained trees between the two methods was identical when the same depth and dataset were used, showing that the obtained trees perform the same over the training dataset.

Summarizing, we can see that the Dynamic programming method is orders of magnitude faster while obtaining the same or comparable trees as the current state-of-the-art and can scale easily to deep trees.

5.4 Bound-based pruning

The following experiments show the number of calls to the depth-two solver, so the number of times the recursive dynamic programming implementation reached the base case. They are counted at increasing depth using both datasets as for the other experiments and are shown in Figure 5a and Figure 5b.

From these experiments, we can see that with deep trees the search space gets pruned, but for smaller trees the difference is negligible. This indicates that there is potential to reduce the search space even further if this work is continued.

5.5 Metric score (total runtime) for training and test datasets

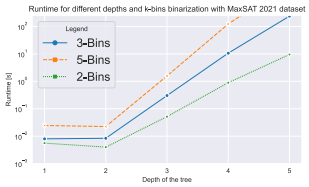
As the two datasets do not contain many instances the parameters β and τ needed to be readjusted and the values of $\beta = 50$ and $\tau = 25$ were chosen.

Here the plot in Figure 6 is showing training and testing total runtime for each instance with the assigned algorithm for the two datasets when the maximum depth of the tree is increased. As we can see the metric's value decreases for the training set, and it stabilizes at around depth 6. The metric over the test set also decreases, but as the depth increases the chance of overfitting also increases and the value fluctuates.

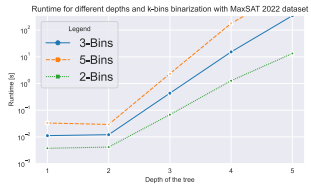
If datasets containing more instances were to be used we could obtain better and more meaningful results, but these preliminary results show that the model performs well and that having more scalable methods that can train deeper trees is important.

5.6 Binarisation strategies

As said in section 5.1 it is possible to discretize the continuous feature in different amounts of bins. The following experiments want to show how increasing this number affects performance, both regarding training runtime and the value of the optimized metric.



(a) Runtime to train trees of increasing depth using different discretization values for MaxSAT 2021



(b) Runtime to train trees of increasing depth using different discretization values for MaxSAT 2022

Figure 7: Experiments to check the runtime of training with increasing depth and different binarization strategies

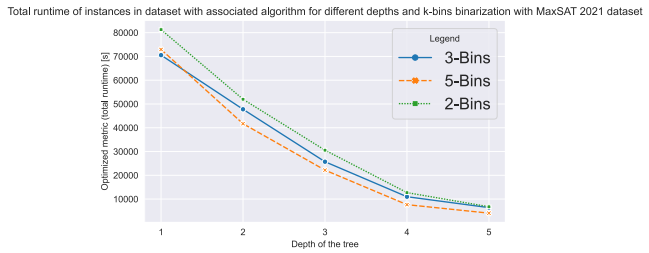


Figure 8: Optimized metric value for trees with increasing depth using different discretization values for MaxSAT 2021

Trees were trained using values of $k = 2, 3, 5$ with depth going from 1 to 5 using the same settings and plot scale as the experiments from subsection 5.3. The runtime needed for training is shown in Figure 7 while the optimized metric value is shown in Figure 8 and Figure 9.

As we can see the runtime grows exponentially for all discretization strategies, but the more information we try to express by using more bins the faster the runtime will increase.

The total runtime that the instances would have using the assigned algorithm still decreases with higher depths, but using more precise encoding for features also increases accuracy. Depending on which is more important a compromise between accuracy and training time needs to be taken into account by users of this method.

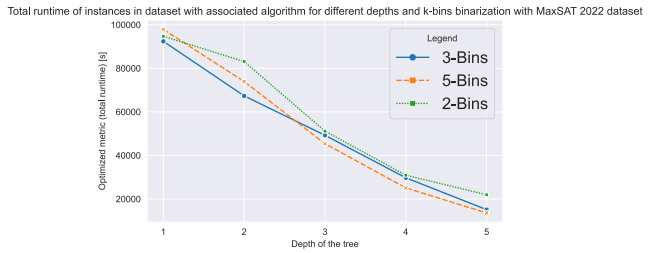


Figure 9: Optimized metric value for trees with increasing depth using different discretization values for MaxSAT 2022

6 Responsible Research

This research and its experiments were conducted under the FAIR principles. These are four main axioms that should be followed to "act as a guideline for those wishing to enhance the reusability of their data holdings" [28] and are listed below together with how they were followed.

- **Findable**: Data and metadata are freely available on the same repository as the code so that they can be easily used to reproduce the same experiments shown in the paper.
- **Accessible**: Access to the data does not require specific authentication, the original results of the MaxSAT competitions are open to the public and so are the processed datasets.
- **Interoperable**: The data is stored in a common CSV format with easy-to-understand explanations shipped together with it.
- **Reusable**: The header and metadata files explain well how the data is structured such that it can be reused for different types of research.

By following these four criteria we hope that the data obtained and used in this research can be used to conduct more experiments and so increase our understanding of algorithms and NP-Hard problems.

7 Conclusions and Future Work

The Algorithm Selection Problem is a relevant problem in Computer Science that could lead to improvements in performance-sensitive environments. Solving it efficiently can not only help make a better choice of algorithm to solve specific problem instances but also give us insights into how different features influence the performance of different algorithms. Multiple machine learning models have been used to solve the ASP, but one of the few that gives us both mathematical guarantees of optimality over the training dataset and an interpretable model is Optimal Decision Trees.

Over the years different heuristics, constraints programming, and Mixed-Integer programming models have been employed to train Optimal Decision trees, but dynamic programming formulations have been shown to increase scalability and reduce runtimes. In this paper, we explored how to build Optimal Decision trees using dynamic programming to solve the algorithm selection problem and showed that this new method outperforms the current state-of-the-art by orders of magnitudes while obtaining the same trees.

In the future, it would be interesting to explore the possibility of stricter upper and lower bounds. The current ones can prune the search space for deep trees, but it is possible to reduce it even more. Moreover, more testing on the actual accuracy and performance of the trees should be investigated more deeply, as in the limited timeframe of this project only datasets of about 500 instances were used to assess scalability and out-of-sample performance. This would require more instances that have accurate measurements over the same algorithms, together with pre-processing and feature extraction for each of these instances. Other datasets not related to the SAT problem might also be used, like for example the freely available MIPLIB[4] or the collection from Dr. Hans D. Mittelmann [19].

[4] https://miplib.zib.de/

The method is promising and is already usable to obtain trees comparable to the current state-of-the-art in a fraction of the time, but more research to assess the accuracy of classification needs to be conducted together with an exploration of how bounds-based pruning can be used to reduce the search space even more than it already does. The optimized metric value, so the total runtime of the instances in the training and test set, is promising and given that it is equal to or better than the work of Boas et al. we can assume that it will perform similarly.

References

[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3146–3153, 2020.

[2] Tomas Balyo, Marijn JH Heule, and Matti Järvisalo. Sat competition 2016: Recent developments. In *Conference on Artificial Intelligence*, pages 5061–5063, 2017.

[3] Kristin P Bennett and Jennifer A Blue. Optimal decision trees. *Rensselaer Polytechnic Institute Math Report*, 214:24, 1996.

[4] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106:1039–1082, 2017.

[5] Dimitris Bertsimas and Romy Shioda. Classification and regression via integer optimization. *Operations research*, 55(2):252–271, 2007.

[6] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[7] Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *The Journal of Machine Learning Research*, 23(1):1169–1215, 2022.

[8] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[9] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[10] Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[11] Frank Hutter, Lin Xu, Holger H Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[12] 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.

[13] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010.

[14] Anthony Karahalios and Willem-Jan van Hoeve. Variable ordering for decision diagrams: A portfolio approach. *Constraints*, 27(1-2):116–133, 2022.

[15] Michail G Lagoudakis, Michael L Littman, et al. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518, 2000.

[16] Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[17] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Non-model-based algorithm portfolios for sat. In *Theory and Applications of Satisfiability Testing-SAT 2011: 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings 14*, pages 369–370. Springer, 2011.

[18] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. In *IJCAI*, volume 13, pages 608–614, 2013.

[19] H.D. Mittelmann. Decision tree for optimization software. http://plato.asu.edu/guide.html, 2021.

[20] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, Joao Marques-Silva, and I Ras. Learning optimal decision trees with sat. *International Joint Conference on Artificial Intelligence 2018*, pages 1362–1368, 2018.

[21] Eugene Nudelman, Kevin Leyton-Brown, Holger H Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. *Principles and Practice of Constraint Programming–CP 2004*, page 438.

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

[23] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[24] Jacobus GM van der Linden, Mathijs M de Weerdt, and Emir Demirović. Optimal decision trees for separable objectives: Pushing the limits of dynamic programming. *arXiv e-prints*, pages arXiv–2305, 2023.

[25] SE Verwer and Yingqian Zhang. Learning decision trees with flexible constraints and objectives using integer optimization. *The 33rd AAAI Conference on Artificial Intelligence*, 2017.

[26] Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. *The 33rd AAAI Conference on Artificial Intelligence*, pages 1625–1632, 2019.

[27] Matheus Guedes Vilas Boas, Haroldo Gambini Santos, Luiz Henrique de Campos Merschmann, and Greet Vanden Berghe. Optimal decision trees for the algorithm selection problem: integer programming based approaches. *International Transactions in Operational Research*, 28(5):2759–2781, 2021.

[28] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

[29] 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.

[30] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2):022022, feb 2019.