Algorithm Selection with Continuous Feature Optimal Decision Trees

An adaption of ConTree's algorithm for instance cost-sensitive classification

Saunaq Chakrabarty
Supervisor: Koos van der Linden, Emir Demirović
[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 22, 2025

Name of the student: Saunaq Chakrabarty
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirovic, Koos van de Linden, Jasmijn Baaijens

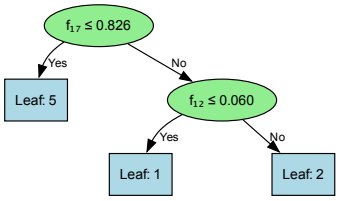
Abstract

Algorithm Selection is a problem that involves finding a way to select the best algorithm out of a portfolio of candidate algorithms, depending on a set of instances for a problem. It has been shown that optimal decision trees that work on binary features are as accurate as state of the art models like random forest, while being more interpretable and smaller, motivating research into alternative, more scalable methods to generate such trees. In this paper we present an optimal decision tree algorithm that operates directly on continuous features and measure its suitability for the algorithm selection problem. We show that our algorithm performs over three orders of magnitude better than other algorithms that build similar decision trees, and that it achieves similar out of sample model selection quality as state of the art methods while being at least 2x faster than similar methods for binary features for higher binarization values.

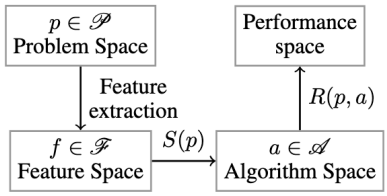
1 Introduction

Although NP-Complete problems are believed to be intractable in the worst-case, it is often possible to solve even very large instances of such problems that arise in practice [5]. However, this may depend strongly on choosing the right algorithm which can make or break the performance of an intractable problem. In many domains, the best choice varies greatly between problem instances, with no algorithm strictly dominating over all problem instances. This is the crux of the Algorithm Selection Problem(ASP). As defined by Rice [18] in 1976, the ASP involves automatically selecting the most suitable algorithm for a given input problem represented by a set of features, from a portfolio of solvers as in 1b. This is typically approached in terms of minimizing(or maximizing) the total metric(or cost). The Algorithm Selection Problem has been studied widely, and many different approaches have been suggested, with several results indicating that random forests, offer the best out of sample performance [5] [14]. However, results also show that they are also not very interpretable [17].

Decision trees are a popular machine learning algorithm for explainable AI because they can capture non-linearities in training data and are also interpretable. A large part of their interpretability is the fact that shallow trees can be easily visualized such as in 1a.



(a) A simple depth two classification tree for arbitrary features



(b) A graph showing the algorithm selection problem

Figure 1

The most commonly used algorithm used to train decision trees is the Classification and Regression Trees algorithm(CART) [8]. This algorithm makes use of a local objective

function that is optimized at each internal node of the tree, and can thus quickly construct the tree. These algorithms have achieved state-of-the-art performance in several machine learning tasks [12], and are implemented in many popular machine learning libraries like scikit-learn [16].

Despite their popularity, decision trees that are created via methods such as CART come with several drawbacks. Primarily, their use of a local objective function comes at the cost of a tree that provably minimizes the training error. Hence, heuristic trees might not represent the data accurately, implying that they do not generalize well to out-of-sample data [6].

This motivated investigation into methods pertaining to the creation of Optimal Decision Trees(ODT), i.e trees that provably maximize a certain evaluation metric, such as accuracy. One issue with this approach is that the problem of finding ODTs is NP-complete [13], with the number of possible trees being upto 9.4×10^8 even when limiting ourselves to a max depth of three for 20 binary features [19].

Nevertheless, a lot of research has been done into this area such as by Demirović et al. [11] and van der Linden et al. [23] that exploit the recursive nature of the tree to train it using dynamic programming, thereby drastically increase scalability. There has also been research into applying these efficient, optimal decision trees in the ASP, showing that such trees are capable of selecting models at a similar quality to state-of-the-art methods such as random forest, while providing significantly more interpretability [17].

A longstanding limitation of ODT algorithms has been their reliance on pre-binarized data. Therefore, these algorithms either use a coarse binarization resulting in loss of optimality; or require a binary feature for every possible threshold on the numeric data, which drastically hurts scalability, because their runtime scales exponentially with the number of such features [9]. This barrier has recently been overcome with the introduction of ConTree [9], marking a key advancement in enabling truly optimal decision trees to natively handle continuous features. For classification problems, we also observe that ConTree was found to have a 0.7% higher test accuracy on out-of-sample data as compared to trees trained with binarization, suggesting an advantage to directly using continuous features [9]. Additionally, ConTree's avoidance of binarization means that it is far more scalable than binary feature ODT methods, as for high binarization values they require far more features.

This naturally motivates us to ask whether we can use the key ideas from ConTree's algorithm to create an algorithm for making optimal decision trees for the ASP, in order to take advantage from the increase in out-of-sample accuracy that comes from using continuous features, as well as the increase in scalability.

Thus we consider the following research question: *How can we create an algorithm that can scalably generate a decision tree that provably minimizes the expected total cost for the ASP while not relying on pre-binarized data?*

The main contribution made by this paper is ConTree++, a novel algorithm that is an adaption of the algorithm presented in ConTree to work with an evaluation metric compatible with the ASP. We introduce a novel lower bound for our metric, as well as different pruning strategies to aid in scalability, in addition to a regularization term to prevent overfitting. We also provide extensive experimentation on 8 datasets comparing our algorithm to state-of-the-art ASP portfolio solvers in terms of out of sample PAR10 score, as well as other optimal decision tree methods in terms of runtime.

Our results show that our method outperform existing methods [7] for constructing optimal decision trees on continuous feature by a factor of over 3 orders of magnitude for trees of depth two, and that trees created by our algorithm have an out of sample algorithm selection quality on par with that of state-of-the-art methods for Algorithm Selection, such as

random forests, while being significantly more interpretable following the results of Poolman et al. [17]. Additionally, we show that our method outperforms ODT algorithms on binary features, by a factor of two, for high binarization numbers, while also obtaining a slightly higher out of sample model selection quality.

2 Related Works

Algorithm Selection: Rice's original framework conceptualized the problem as a mapping from a feature space of problem instances to an algorithm space, where the goal is to optimize a performance metric (e.g., runtime, accuracy, or memory usage) [18]. This abstract formulation emphasized the importance of understanding the relationship between problem instance features and algorithm performance.

In the early days, research primarily focused on single domain applications such as finding the best SAT solver. Portfolio-based methods such as SATzilla [25] performed incredibly well at this, and won several gold medals in SAT competitions [17], thereby motivating research into portfolio based algorithm selection in other domains.

Subsequent research expanded ASP techniques to other domains, including Constraint Satisfaction Problems (CSP), Planning, and Answer Set Programming. Notable systems include SUNNY [2], which solves the ASP for CSP and is based on k-NN, and Claspfolio, which solves the ASP for Answer Set Programming and uses several mechanisms such as k-NN, random forests, and regression [17].

The advent of ASlib [5] in 2016 marked a significant advancement by providing a standardized benchmark library for cross-domain evaluation of algorithm selection techniques.

Recent approaches to the algorithm selection problem involve the use of graph-convolutional network-based genetic adversarial networks, and graph neural networks, which have been applied to the TSP [21]. Cross-domain studies on the ASP still seem to show that smaller, simpler models such as random forest tend to outperform more complex models [14] [5]. Nevertheless, the shift in interest towards more complex, black-box models in solving this problem is noteworthy, and it further motivates the need to develop lightweight, simple and interpretable models for algorithm selection.

Optimal Decision Trees: The construction of Optimal Decision Trees was shown to be NP Hard by Hyafil et al. [13], establishing that finding the smallest decision tree consistent with a given dataset is computationally intractable in the general case.

Some initial approaches on finding ODTs include the use of Extreme Point Tabu Search [3], which involves fixing the structure of the tree and then solving a system of linear inequalities using existing optimizers. Additionally there were many approaches done through the use of Mixed-Integer-Programming(MIP) and constraint programming(CP) such as by Bertisimas et al. [4] and Verwer et al [24].

In more recent times, there have been several approaches involving dynamic programming and Branch and Bound search, that exploit the recursive nature of the tree. These approaches sacrifice generalizability to gain better runtime performance [20]. Some examples of these include DL 8.5 by Aglin et al. [1] and MurTree by Demirović et al. [11]. Of the works stated above MurTree can obtain optimal classification trees and has been further generalized to any optimization task in STreeD [22]. These methods find optimal trees, while achieving orders of magnitude better scalability than the prior MIP methods.

The previous approaches discussed do not directly handle continuous features. Instead they require a course binarization of the feature space, which affects optimality, or a binary variable at each threshold which hurts scalability. ConTree [9] and QuantBnB [15] are two

very recent examples of algorithms that are capable of handling continuous features directly, and are treated as the starting point of this work, which is aimed at extending ConTree to handle instance-cost sensitive classification.

Optimal Decision Trees for Algorithm Selection: Optimal decision trees as a method for Algorithm Selection, have been visited before by Vilas et al. who built a MIP model to train optimal decision trees [7]. They were shown to provide results similar to that of random forest regressors, and the authors provided ample evidence against overfitting.

Unfortunately, they were limited in that they only had 1000 problem instances for their whole dataset, and they had significant difficulties with scalability. They found that optimal or near optimal solutions were only found for a subset of upto 200 instances. For larger datasets the solution timed out, or returned suboptimal values.

Another approach to applying Optimal Decision Trees was presented by Segalini et al. [20], who adapted the STreeD framework for per instance cost sensitive applications [11] and found that optimal decision trees can be computed much more scalably. Unfortunately this study was only limited to MaxSAT data and lacked analysis on out-of-sample accuracy, as the main focus was in improving scalability.

Poolman et al. [17] presented a comprehensive evaluation on the performance vs interpretability tradeoff for optimal decision trees. They found that the instance cost sensitive(ICSC) STreeD algorithm provided a similar performance to the state-of-the-art random forest regressor, [5], while providing substantially more interpretability. This makes research into such decision trees that work on continuous features promising, due the performance increase that such trees are known to have [9].

3 Preliminaries

In this section we introduce notation, formally describe the ASP, introduce our evaluation metric and briefly describe the notation we use for our algorithm, that has been taken from ConTree.

3.1 Algorithm Selection

The Algorithm Selection problem can be formalized as follows [18]:

- **The Problem Instance Space ($\mathcal{P}$):** The set of problem instances where $p \in \mathcal{P}$ that we consider.
- **The Algorithm space ($\mathcal{A}$):** The set of algorithms which can be used to solve the problems in $\mathcal{P}$.
- **Performance metric ($\mathcal{M}$):** a function $m : \mathcal{P} \times \mathcal{A} \to \mathbb{R}$ that measures the performance of algorithm a on problem x. This is the metric that needs to be minimized (or maximized).
- **Feature Space ($\mathcal{F}$):** The set of all features that describes all instances of the considered problem.

The goal of the problem is to find a mapping s such that:

$$\sum_{p \in \mathcal{P}}^{n} \mathcal{M}(p, s(p))$$

is minimized. There are many approaches to find such a mapping. One such interpretation is that of a *regression* problem, by considering the runtime data of each individual algorithm as a training data for a regression algorithm. In this approach we take the algorithm with the lowest predicted runtime as the prediction. This approach is used by random forest regressors, discussed later. In this paper we interpret the ASP as an *instance cost-sensitive classification* problem. This type of problem is similar to classification but considers different misclassification costs for each label and each instance. This is especially relevant for a problem like the ASP where runtimes can differ wildly between algorithms and problem instances.

A common evaluation metric for a portfolio selection algorithm in the context of the ASP is the PAR10 score which is defined as follows:

$$PAR_{10}(s) = \frac{\sum_{p\in\mathcal{P}}^{n} \mathcal{M}^{'}(p, s(p))}{|\mathcal{P}|}$$

where

$$\mathcal{M}^{'}(p,s) = \begin{cases} \mathcal{M}(p,s) & \text{if } \mathcal{M}(p,s) \leq T \\ 10T & \text{else} \end{cases} \tag{1}$$

Where T refers to the value at which the algorithms in the dataset time out.

In order to effectively compare PAR10 scores across different datasets, which may have different timeouts and different average runtimes, we need to normalize the PAR10 score [17]. We do this by comparing it to the Virtual Best Solver(VBS) and the Single Best Solver(SBS). The VBS is the best solver for a particular instance and the SBS is the best solver on average across the whole dataset.

$$NormalizedPAR_{10}(s) = 1 - \frac{PAR_{10}(s) - VBS}{SBS - VBS} \tag{2}$$

We want to maximize this score, which involved minimizing the raw PAR10 score.

3.2 ConTree

The algorithm introduced in this paper is an adaption of the ConTree algorithm [9]. Hence, the notation used is the same and is summarized in brief below.

Notation: The input to our algorithm is a dataset $\mathcal{D}$ with $n = |\mathcal{D}|$ observations. Each observation can be represented by (x, a) where $x \in \mathbb{R}^p$ where p is the number of features($|\mathcal{F}|$) and $a \in \mathbb{R}^q$, where q is the number of algorithms($|\mathcal{A}|$). $F = \{f_1, f_2, f_3 ... f_p\}$ is the set of all features in our problem and is the first p columns in the dataset. $\mathcal{D}^f$ represents one of these such columns, and includes every instance (x, a) in $\mathcal{D}$, sorted by their values in the column f. U^f represents all the unique sorted values in $\mathcal{D}^f$, with the instances having a duplicate value in f being removed. Similarly we define $S_f = \{\frac{U_1^f + U_2^f}{2}, \frac{U_2^f + U_3^f}{2} ..., \frac{U_{k-1}^f + U_k^f}{2}\}$ where $k = |U^f|$. S_f is the set of thresholds to be considered on the feature f for that dataset. $z(\tau)$ refers to the index of the largest element in D^f where the feature value $x_f \leq \tau$. Similarly, $u(\tau)$ refers to the index of the largest element in U^f where the feature value $x_f \leq \tau$. We use $\mathcal{D}(f \geq \tau)$ to describe the subset of all observations (x, a) such that $x_f \geq \tau$ and vice-versa for $\mathcal{D}(f \leq \tau)$. We also define the quantity $w(\tau)$ which is defined as: $w(\tau) = \sum_{(x,a)\in D(f\leq\tau)}(max(a))$. Finally, we also work with the new objective function J that is the total cost incurred by using our decision tree on a dataset of problems. $J_{\mathcal{D}}$ refers to the total metric incurred, when building a classifier for the dataset $\mathcal{D}$, and J_τ refers to

the total metric incurred when building a decision tree that splits at the threshold τ, with $J_{\tau R}$ and $J_{\tau L}$ being the left and right total metric of splitting on τ respectively.

ConTree used 3 novel pruning techniques., namely Neighbourhood Pruning(NB), Interval Shrinking(IS) and SubInterval Pruning(SP). Each of these 3 techniques as well as the Depth Two solver, are specific to the objective function used in ConTree, which is the misclassification score. Each of these techniques, are re-implemented and proven in order to transfer them over to this implementation of ConTree.

4 ConTree++

Problem Definition: The problem statement of this algorithm is essentially the same as the ASP, with some caveats. If $\mathcal{T}(\mathcal{D}, d)$ represents the set of all decision trees of max-depth d on the dataset $\mathcal{D}$, then we need to find the tree t_{opt} such that:

$$t_{opt} = \arg\min_{t} \sum_{(x,a)\in\mathcal{D}} \mathcal{M}(x, t(x)) + \lambda(SBS - VBS)N_l$$

where N_l represents the number of leaf nodes in t. This algorithm, like ConTree and STreeD is limited to binary axis-aligned trees: every branching node splits on precisely one feature $f \in \mathcal{F}$ based on a threshold τ such that every observation with $x_f \leq \tau$ goes left in the tree while the rest goes to the right.

We prevent overfitting by adding a regularization term to penalize the size of the tree, which is represented by the number of leaf nodes. This regularization score is scaled up by the normalization factor $SBS - VBS$ as the leaves contain only the raw PAR10 score, however it is useful to have our regularization parameter λ normalized as our ultimate evaluation metric is normalized for effective cross-domain evaluation.

4.1 Main Algorithm

Much like ConTree, this algorithm constructs an ODT by recursively performing splits on every branching node within a full tree of pre-defined depth. Subproblems are identified by the dataset $\mathcal{D}$ and the remaining depth limit d. This results in the following recursive DP formulation.

$$CT^{++}(\mathcal{D}, d) = \begin{cases} \arg\min_a \sum_{(x,a)\in\mathcal{D}} \mathcal{M}(x, a) + \lambda(SBS - VBS) & \text{if } d = 0 \\ \min_{f\in\mathcal{F},\tau\in S^f}(CT^{++}(\mathcal{D}(f \leq \tau), d-1) + CT^{++}(\mathcal{D}(f > \tau), d-1)) & \text{else} \end{cases}$$

Given a splitting feature f , computing the total cost J_τ for all possible split points $\tau \in S^f$ is computationally expensive since each split point considered requires solving two (potentially large) subproblems. ConTree fixes this problem by proposing three novel runtime pruning techniques [9]. We adapt these and prove their validity for the new objective function. Additionally, the Depth Two solver is also adapted for the new objective function and provides a significant speedup to the algorithm.

4.2 Similarity Lower Bound:

ConTree makes use of the Similarity Lower Bound proposed in MurTree by Demirović et al. [11], which is based on the misclassification score. We adapt it for our new metric to make

it the following:

$$J_{D_{new}} \geq J_{D_{old}} - \sum_{(x,a) \in D_{out}} max(a)$$

where $D_{out} = D_{old} \backslash D_{new}$.

Proof: The proof follows in a similar fashion to the proof of the original Similarity Lower Bound in Demirović et al. [11]. Consider $D_{in} = D_{new} \backslash D_{old}$, $D_{out} = D_{old} \backslash D_{new}$ and $D_{same} = D_{out} \cap D_{in}$. We observe that removing D_{out} from D_{old} may reduce the total cost by at most $A = \sum_{(x,a) \in D_{out}} max(a)$. This is under the assumption that the classifier predicts the worst possible algorithm on every instance in D_{out} and its predictions on D_{same} remain unchanged. Therefore:

$$J_{D_{old}} - J_{D_{old} \backslash D_{out}} \leq A$$

$$\implies J_{D_{old}} - A \leq J_{D_{same}}$$

Another key observation is that adding more observations to D_{same} cannot reduce the total cost in the worst case, as in this case, all new observations are misclassified so they add something to the cost, and the instances in D_{same} remain the same. Hence:

$$J_{D_{new}} \geq J_{D_{same}}$$

$$\implies J_{D_{old}} - \sum_{(x,a) \in D_{out}} max(a) \leq J_{D_{new}} \tag{3}$$

4.3 Pruning Techniques

Based on this new similarity lower bound, 3 separate pruning techniques are proposed similar to ConTree. Like ConTree, the key idea of the algorithm is that the solution when splitting on a feature f on a threshold τ provides a lower bound for future calls on the same feature, with a different threshold. Unlike ConTree, the new objective function makes it so that the indices that can be pruned, cannot be removed in constant time like in ConTree, as every individual instance contributes a different amount to the lower bound, depending on what the maximum runtime of each instance is. To this end, this paper presents a novel method of finding the correct index to split on, in logarithmic time.

Theorem A: *If $\mathcal{UB}$ is the best existing score needed, and if J_τ is an already computed solution to the total metric of an optimal decision tree on a threshold τ, then no threshold $\tau^{'}$ with $\sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a) < J_\tau - \mathcal{UB}$ or $\sum_{(x,a) \in \mathcal{D}(\tau \geq f > \tau^{'})} max(a) < J_\tau - \mathcal{UB}$ can provide an improving solution*

Proof: This follows directly from the similarity lower bound. Consider the case where $\tau^{'} > \tau$. In this case, $\mathcal{D}(f \leq \tau) \subset \mathcal{D}(f \leq \tau^{'})$. Therefore,

$$J_{\tau L} \leq J_{\tau^{'} L}$$

as here $\mathcal{D}_{old} \backslash \mathcal{D}_{new} = \emptyset$. Additionally for the right side, we have

$$J_{\tau R} - \sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a) \leq J_{\tau^{'} R}$$

Because $\mathcal{D}(f > \tau) \backslash \mathcal{D}(f > \tau^{'}) = \mathcal{D}(\tau^{'} \geq f > \tau)$ Adding these two together we get

$$J_{\tau'} \geq J_{\tau} - \sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a)$$

In order for $J_{\tau'}$ to be an improving solution it has to be less than $\mathcal{UB}$. Meaning:

$$\mathcal{UB} \geq J_{\tau} - \sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a)$$

$$\implies \sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a) \geq J_{\tau} - \mathcal{UB} \tag{4}$$

(4) must hold in order to be an improving solution. Contrapositively, any value of $\tau^{'}$ for which the following holds:

$$\sum_{(x,a) \in \mathcal{D}(\tau^{'} \geq f > \tau)} max(a) < J_{\tau} - \mathcal{UB}$$

cannot be an improving solution. The proof follows similarly for values of $\tau^{'} < \tau$

Corollary A: *Consider $\Delta = J_{\tau} - \mathcal{UB}$. If τ_1 is the smallest threshold such that $\sum_{(x,a) \in \mathcal{D}(\tau_1 \geq f > \tau)} max(a) \geq \Delta$ and if τ_2 is the largest threshold such that $\sum_{(x,a) \in \mathcal{D}(\tau \geq f > \tau_2)} max(a) \geq \Delta$ then the only indices that need to be checked are those indices $\tau^{'}$ such that either $z(\tau^{'}) \leq z(\tau_2)$ or $z(\tau^{'}) \geq z(\tau_1)$*

Like ConTree, we keep track of a set of intervals of indices with possible candidates for an improving solution. After each pruning strategy, we reduce the total size of the intervals considered, until we eventually settle on one optimal solution.

Neighbourhood Pruning: After a split point J_u is computed on a split index u with threshold value τ, we can make use of Corollary A to prune away suboptimal solutions. To this end, we need to compute the values τ_1 and τ_2. This may be naively done by going over the array in one pass. However, we can precompute the prefix sum(or cumulative sum) of the worst case algorithm across the dataset sorted by each feature($w(\tau)$) and store it as a field in each problem instance. With this value precomputed, we can compute τ_1 and τ_2 in logarithmic time by using a binary search to find the smallest value that is greater than $w(\tau) + \Delta$ and the largest value that is smaller than $w(\tau) - \Delta$ respectively. Afterwards we can proceed with the same procedure as in ConTree, by implementing the functions $\overline{A}(u, \Delta) = \{u^{'} \in [m] | U^{f}_{u'} < \mathcal{D}^{f}_{\tau_1}\}$ and $\underline{A}(u, \Delta) = \{u^{'} \in [m] | U^{f}_{u'+1} > \mathcal{D}^{f}_{\tau_2}\}$ which also work in logarithmic time. These are used to convert from the datapoint index in $\mathcal{D}^{f}$ to the unique value index, as these are the indices that we can split upon. These metrics can be used to create a new set of intervals to check, with the following function

$$P_{NB}([i, j], u, \Delta) = \{[i, \underline{A}(u, \Delta)], [\overline{A}(u, \Delta), j]\}$$

Interval Shrinking: Interval Shrinking is simply a lazy evaluation of neighbourhood pruning, and makes use of Theorem B

Theorem B: *If w is a split point with a precomputed total metric of J_w such that the left total metric $J_{wL} = 0$ then any split point $u < w$ will have a total metric $J_u \geq J_w$. Similarly, if we have a point with $J_{wR} = 0$ then any split point $v > w$ will have a total metric $J_v > J_w$.*

Proof: First we consider only the left side. suppose $u < w$ and $J_{wL} = 0$. Then $\mathcal{D}(f \leq S_u^f) \subset \mathcal{D}(f \leq S_w^f)$ and $\mathcal{D}(f > S_w^f) \subset \mathcal{D}(f > S_u^f)$. Note also that $J_w = J_{wR}$. Following the similarity lower bound, $J_{uR} \geq J_{wR}$ as $\mathcal{D}_{out} = \mathcal{D}(f > S_w^f) \backslash \mathcal{D}(f > S_u^f)$ is empty. Additionally, we also know that $J_{uL} \geq 0$ as the total metric is assumed to be non-negative. Therefore, combining these two statements we get, $J_{uR} + J_{uL} = J_u \geq J_{wR} = J_w \implies J_u \geq J_w$.

The proof follows similarly for the right side.

We use this theorem for pruning the search space in a similar fashion to ConTree by using the following expression:

$$P_{IS}([i,j], u, v, \Delta_u, \Delta_v, M_L, M_R) = \{max(i, M_L + 1, \overline{A}(u, \Delta_u)), min(j, M_R - 1, \underline{A}(v, \Delta_v))\}$$

Here u, v are the previously computed split indices, $\Delta_u = J_u - \mathcal{UB}$, $\Delta_v = J_v - \mathcal{UB}$ and M_L, M_R are the indices of the leftmost and rightmost index at which a tree of total metric zero has been found.

Sub-interval Pruning: Subinterval Pruning makes use of the following theorem to prune away suboptimal solutions.

Theorem C: *If [i,j] is our current interval, and $u < i$ and $v > j$ are two split indices for which the quantities $J_{uL}, J_{uR}, J_{vL}, J_{vR}$ have already been computed, then if $J_{uL} + J_{vR} > \mathcal{UB}$, then no $w \in [i, j]$ can provide a better total metric on $\mathcal{UB}$.*

Proof: We know that $w \geq i > u$ and $w \leq j < v$. Following the similarity lower bound, $J_{wL} \geq J_{uL}$ and $J_{wR} \geq J_{vR}$. Combining these two we get: $J_w = J_{wL} + J_{wR} \geq J_{uL} + J_{vR}$. This means that if $J_{uL} + J_{vR} > \mathcal{UB}$, then $J_w > \mathcal{UB}$, meaning w cannot be an improving solution.

Hence we can define the pruning function for Sub-interval Pruning as:

$$P_{SP}([i,j], \mathcal{UB}, J_{uL}, J_{vR}) = \begin{cases} \emptyset & \text{if } J_{uL} + J_{vR} \geq \mathcal{UB} \\ [i,j] & \text{else} \end{cases}$$

4.4 General Case Solver

The main loop of the algorithm proceeds identically to ConTree. The pseudo code for this algorithm is presented in appendix A as algorithm 2.

The main loop of our algorithm and ConTree involves going over every feature f and running Algorithm A to find the optimal decision tree if we decide to split on that feature. Algorithm A tries to find the optimal threshold to split on for each feature.

It does so by maintaining a queue of intervals. At each step an interval is popped out and subjected to sub-interval pruning and interval shrinking, which reduce the size of the interval, or possibly avoid having to consider it all together. After this, the point at the middle is considered and an optimal decision tree of depth $d - 1$ is computed by splitting on this threshold. Depending on the result of this computation, neighbourhood pruning is then applied to the resulting interval which is split into two smaller intervals.

If the depth of the next tree to be computed is two, the algorithm will call the specialized solver method that implements another procedure adapted from ConTree, which computes optimal decision trees of depth two very efficiently. This procedure is described in more detail in the section below. Otherwise, it will first compute the left subtree of the dataset if we split at w on the current feature, and then use the result of the left subtree as a tighter upper bound for the right subtree. The right subtree is only computed if the upper bound indicates that it may be an improving solution. We also use ConTree's adaptive procedure for a tighter upper bound on the right subtree.

Finally the algorithm terminates once all of the intervals have been searched and the queue is empty. Additionally if a tree with a perfect total metric of zero is found, then the search will terminate earlier.

4.5 Depth Two Solver

The Depth Two Solver is a very important subroutine for DP based ODT algorithms, with its use in ConTree being responsible for reducing runtimes by a factor of 320 [9].
In this paper, we present an adaptation of ConTree's Depth Two solver that is capable of computing Optimal Decision Trees that operate on the new objective function of the total metric and also does not run into memory errors or issues.

Algorithm 1 D2$(\mathcal{D}, f, w)$

$J_L \leftarrow |\mathcal{D}|$, $J_R \leftarrow |\mathcal{D}|$
for $(x, a) \in \mathcal{D}$ **do**
    $FQ_L^a \leftarrow FQ_L^a + a$
$FQ_R^a \leftarrow (\sum_{(x,a)\in\mathcal{D}} a) - FQ_L^a$
for $f_2 \in \mathcal{F}$ **do**
    $C_L^a \leftarrow 0$, $C_R^a \leftarrow 0$
    for $(x, a) \in \mathcal{D}$ sorted by f_2 **do**
        if $x_{f_1} \leq S_w^{f_1}$ **then**
            $J_{LL} \leftarrow \min_{\hat{a}} C_L^a$
            $J_{LR} \leftarrow \min_{\hat{a}}(FQ_L^a - C_L^a)$
            if $J_{LL} + J_{LR} \leq J_L$ **then**
                $J_L \leftarrow J_{LL} + J_{LR}$
            $C_L^a \leftarrow C_L^a + a$
        else
            same procedure for the other side
        if $J_L + J_R = 0$ **then**
            break
Output: (J_L, J_R)

Algorithm B shows the procedure used to compute an optimal decision tree of depth two if we split at the point w. As mentioned above this algorithm is only called as a subroutine in the main loop of the algorithm discussed above, where f and w have already been decided. The main point of the algorithm is that at the beginning, we first precompute the variables FQ_L^a and FQ_R^a which counts up the contribution of algorithm a for each datapoint that is put into the left and right subtree respectively, if we split on w for feature f_1. Then in order to select the second feature f_2, we loop over all of them and on each of them, go over each threshold and keep track of the variables C_L^a and C_R^a which count the contribution of each algorithm to the total metric, for all the datapoints on the left subtree of f_2 on the left or right subtree of f_1 respectively. Then the values of the total metric of the right subtree on f_2 for both branches of f_1 can be easily computed with the following formulae:

$$C_{LL}^a = C_L^a \qquad C_{LR}^a = FQ_L^a - C_L^a$$
$$C_{RL}^a = C_R^a \qquad C_{RR}^a = FQ_R^a - C_R^a$$

The optimal total metrics can then easily be calculated by simply finding the label that is minimizes the value of the total metric for each subtree. This algorithm operates in $O(|\mathcal{D}||\mathcal{F}|$.

5 Experimental Setup and Results

Our experiments and results aim to answer the folowing questions:

- What is the effect of the pruning techniques on runtime?
- How does our algorithm compare to other algorithm for the ASP using ODTs in terms of runtime?
- How does our algorithm compare to state-of-the-art methods for the ASP in terms of algorithm selection quality?
- How do ODTs made on continuous features compare to those made on binary features, in terms of both tree quality and scalability/runtime?

To this end we proposed and performed the following experiments: Experiment 1 compares the runtimes of our algorithm with a max depth of three with different combinations of pruning active, showing the combination of NB and SP is optimal. Experiment 2 will compare the runtime of our algorithm to the MIP model by Boas et al.[7], confirming our method's superiority in terms of scalability and runtime on trees of depth 2, as the MIP model times out on all of these datasets. Experiment 3 will compare the out-of-sample normalized PAR10 score of the optimal decision trees generated by our algorithm with random forest regressors showing that we have comparable performance, and finally experiment 4 will compare our algorithm to the STreeD instance cost sensitive classifier(ICSC), which acts on binarized features, in terms of out-of-sample PAR10 score, as well as training time depending on the number of binarizations, where we find that our methods severely outperforms binarizations of high value.

5.1 Setup and Data Preparation

Data Source: We chose to perform our experiments on 8 data sets from the ASLib dataset for algorithm selection. ASLib is a repository of data collected, specifically for the purposes of the Algorithm Selection problem. It covers a variety of different problem domains including SAT, QBF(quantified boolean formulas), CSP(constraint programming), ASP(Answer set programming) and many more.

ASLib removes the need for us to precompute the features and runtimes for the algorithms ourselves. It additionally makes it much easier for other reseachers to compare their work against our results in the future.

We only choose the datasets wherein the metric being measured is runtime or directly a PAR10 score, so as to ensure the problem remains a minimization problem. Additionally, we also tried to choose datasets from a wide range of problems, so as to confirm the cross-domain applicability of our method.

Data Preprocessing: We remove feature columns that have the same value across all instances. Additionally, we also subtract the metric of the best performing algorithm from the metrics of all of the other ones. This ensures that the best runtime of a decision tree has a tight lower bound of zero, which greatly speeds up performance, as mentioned

Table 1: Summary of Scenarios

Scenario Name	# **Feat.**	# **Algo.**	# **Inst.**
ASP-POTASSCO	134	11	1077
CSP-2010	69	2	1695
BNSL-2016	86	8	1179
CPMP-2015	22	4	527
GLUHACK-2018	48	8	353
MAXSAT-2012PMS	30	6	747
PROTEUS-2014	193	22	678
QBF2016	46	24	813

Table 2: Hyperparameter Ranges for Models

Model	**Hyperparameters**
STreed Instance Cost Sensitive	max depth $\in \{2, 3, 4, 5\}$ max num nodes $\in \{3, 5, 10, 15, 20, 25, 31\}$ bins $\in \{2, 3, 5\}$
ConTree++	max depth $\in \{2, 3, 4\}$ reg. score(λ) $\in \{0.01, 0.05, 0.1, 0.005, 0\}$
Random Forest Regressor	num estimators $\in \{100, 200, 300\}$ max depth $\in \{None, 3, 4, 5, 10, 20, 50\}$ min split $\in \{2, 5\}$ min leaf $\in \{1, 2\}$

earlier. Additionally when using STreeD for this problem, we require feature binarization, which requires some extra steps. For each feature in the dataset, the following precedure is followed:

- Sort the feature column ascendingly.
- Divide these sorted features into a number of equally spaced bins. The number of bins is provided as a hyperparameter.
- Identify the thresholds that identify these bins by looking at the position of feature elements in the sorted list.
- For each threshold, set all of the elements lesser than it to 1 and set all of the elements greater than it to 0.

This procedure has the number of bins as a hyperparameter, hereafter referred to as the binarization value.

Model Selection and Tuning: In order to adequately assess the strength of our algorithm, we compare it to algorithms along different evaluation metrics. For experiment 2, the only existing models for building instance cost sensitive ODTs for the ASP directly on continuous features that we know of is the MIP model by Boas et al. [7], so we only include this model as a point of comparison. For experiment 3 we made our selection from the limited set of algorithms and models that are equipped to handle cross-domain algorithm selection.

We chose to focus on tree based models such as random forests, as they have been shown to have the highest out of sample PAR10 score for the ASP [17] [14]. The implementation of these models were the same as in the original ASLib paper by Bischl et al. [5], which made use of the implementations provided in scikit-learn. There are also other methods for the algorithm selection problem that use complicated, black box models such as transformers. However, results show that their performance does not differ all that much from random forests, despite being much less interpretable [14]. Hence we chose to omit them from this comparison, as the random forest is a more interesting comparison to optimal decision trees.

We chose to train the models by tuning the hyperparameters of all six models, using the provided parameter ranges, with grid search and a nested cross-validation setup (with ten internal folds as specified by ASlib) for each scenario. A further specification of these ranges

can be found in Table 2. This is done to ensure both unbiased performance results and to select the best model within the parameter ranges for each scenario.

Experiments 1 and 3 are run on the DelftBlue supercomputer [10], which is equipped with Intel XEON E5-6448Y 32C 2.1GHz CPUs. Each experiment is configured with 64 CPU cores and 2GB of RAM per core. Experiments 2 and 4, and the results in Figure 1, are run on an Alienware M15R5 with an Intel i7 2.1 GHz processor, 22 cores and 16GB of RAM total.

5.2 Results and Discussion:

The results of the MIP model are ommitted as the MIP model times out on all of the datasets, where as our algorithm does not. For the comparison with MIP and the pruning analysis, we set a time out of 1500 seconds. The results in Table 3 clearly shows that the

Table 3: Runtime(seconds) for trees of depth three with different pruning setups, - denotes a timeout.

Scenario Name	**All**	**NB**	**SP**	**NB+SP**
ASP-POTASSCO	1095	-	-	**1067**
CSP-2010	**44**	-	70	46
BNSL-2016	265	-	279	**259**
CPMP-2015	**2**	**2**	3	**2**
GLUHACK-2018	32	-	56	**32**
MAXSAT-2012PMS	**5**	-	6	**5**
PROTEUS-2014	1286	-	-	**1266**
QBF2016	**421**	-	454	**421**

Table 4: Out of sample PAR10 between ConTree, Random Forest and STreeD, normalized according to equation (2)

Scenario Name	**ConTree++**	**Random Forest**	**STreeD**
ASP-POTASSCO	0.76	**0.84**	0.76
CSP-2010	0.88	0.82	**0.94**
BNSL-2016	0.77	**0.86**	0.78
CPMP-2015	0.19	**0.36**	0.29
GLUHACK-2018	**0.28**	0.17	0.19
MAXSAT-2012PMS	**0.90**	0.89	0.87
PROTEUS-2014	0.68	**0.70**	**0.70**
QBF2016	**0.58**	0.41	0.5

combination of Neighbourhood Pruning and Subinterval Pruning provides the best average runtime(387.25s), even compared to applying Interval Shrinking as well(393.76s). This can be explained by the overhead of computing left and right thresholds in logarithmic time, exceeding the pruning strength of the method. Hence, in the other experiments of this paper, only the combination of **NB+SP** is used. Overall we see that the most important pruning technique in terms of runtime is Subinterval Pruning. The addition of Neighbourhood Pruning to SP provides a 25% speedup(geometric mean), whereas the addition of Subinterval Pruning to Neighbourhood Pruning provides a 8×(geometric mean) speedup.

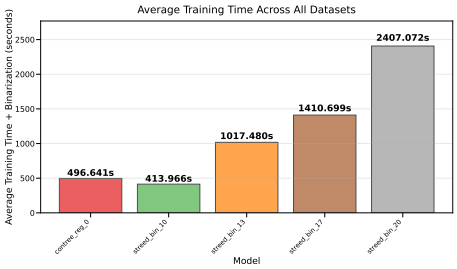
For depth 2 trees, our method drastically outperforms the Boas et al.'s MIP model on all datasets. Our method has a maximum runtime of 10 seconds, meaning it provides a runtime speed up of at least three orders of magnitude. This is consistent with their findings in the original paper, as they were having issues with scalability on datasets of merely 37 features and 200 instances, whereas the datasets that we are consider have either far more instances, or far more features. Additionally, they worked with a much higher timeout (4000s), as compared to ours(1500s). For higher depth trees, we imagine the a similarly high speedup, as the search space for optimal decision trees is exponential with the max depth.

In terms of the out of sample PAR10 score, we observe that all three models are evenly matched. ConTree++ and Random Forest both have slightly higher average scores(0.63)

than STreeD(0.62) illustrating as slight advantage to directly using continuous features over binarizing them. This is consistent with the result from ConTree [9] that found a 0.7% higher out of sample accuracy than STreeD for classification.

Figures 2a & 2b illustrate the scalability of our method with that of STreeD at different binarizations. We observe exponentiality with regards to the number of features, with QBF2016 being a notable outlier, possibly due to its high number of algorithms. We observe that our method has a lesser average runtime across datasets than STreeD with a binarization value greater than 13, outperforming such models by at least a factor of 2 in runtime, indicating our method's superiority in scalability in comparison to high binarization values(>=13).

(a) Average training time by dataset(s), averaged over 3 runs, datasets sorted by number of features



(b) Average training time across all datasets per model

Figure 2: Scalability comparison between STReeD and ConTree

6 Conclusions and Future Work

We have clearly illustrated that making use of dynamic programming and ConTree's algorithm it is finally possible to scalably compute ODTs on continuous features, for algorithm selection, with our method clearly outperforming existing methods for this task. We accomplished this by modifying ConTree's pruning techniques and depth two solver for the PAR10 score, an evaluation metric commonly used for the ASP.

Additionally, our results show that our method performs similarly to state-of-the-art methods like Random Forest on out of sample selection quality, while providing a much greater deal of interpretability and a much smaller, more space efficient model following Poolman et al.'s experiments[17]. Apart from a small advantage in out of sample selection quality, our results also show that our method outperforms STreeD in scalability for larger binarization values, meaning that for datasets in which a high number of bins is required for binarization, ConTree++ is a considerably faster choice.

Future research into this topic could involve improving the scalability of this method by introducing tighter lower bounds and additional pruning techniques. Finally, experimentation into the interpretability from models generated via ConTree++ and Random Forest, on a larger dataset could be explored.

7 Responsible Research

This paper adheres to the FAIR principles to ensure that the data and code used in the experiments are:

- **Findable:** Metadata and related content are on a public repository, making it easy for researchers to locate and access.
- **Accesible:** The code for the algorithm as well as the scripts involved in the experiment setup, along with the datasource is public, making it easy to replicate.
- **Interoperable:** The code is designed to run on Windows and Unix systems, facilitating easier use by the research community.
- **Reusable:** The code contains all the necessary details and instructions on how to replicate the experiments, thereby promoting code reuse and further research based on this paper.

LLMs were used to to aid in writing experiment scripts, as well as in generating the relevant visuals and tables. It was also used to aid in correctly formatting tables and figures in LaTeX

A Appendix

A.1 Pseudocode for main loop of ConTree++

Algorithm 2 Branch($\mathcal{D}, d, f, \mathcal{UB}$)

$J_{opt} \leftarrow min_{\hat{a} \in \mathcal{A}} \frac{\sum_{(x,a) \in \mathcal{D}} \mathcal{M}'(x,a)}{|\mathcal{D}|}$
$M_L \leftarrow 0, M_R \leftarrow \Updownarrow + 1$
$Q \leftarrow [1, m], \mathcal{V} \leftarrow \emptyset$
while $|Q| > 0$ **do**
  $[i, j] \leftarrow Q.pop()$
  $u, v \leftarrow B(i, j), \mathcal{V})$
  $\Delta_u \leftarrow J_u - \mathcal{UB}, \Delta_v \leftarrow J_v - \mathcal{UB}$
  $[i, j] = P_{SP}([i, j], \mathcal{UB}, J_{uL}, J_{vR})$
  $[i, j] = P_{IS}([i, j], u, v, \Delta_u, \Delta_v, M_L, M_R)$
  if $|[i, j]| = 0$ **then**
    continue
  $w \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if $d = 2$ **then**
    $J_{wL}, J_{w,R} \leftarrow D2(\mathcal{D}, f, w)$
  else
    $\mathcal{D}_L \leftarrow \mathcal{D}(f \leq S_w^f), \mathcal{D}_R \leftarrow S_w^f$
    $J_{wL} \leftarrow CT^{++}(\mathcal{D}_L, d-1, \mathcal{UB})$
    $\eta \leftarrow min(z(w) - z(i), z(j) - z(w))$
    $\mathcal{UB}_R \leftarrow max(\mathcal{UB} - J_w, \eta)$
    if $\mathcal{UB}_R \leq 0$ **then**
      $J_{wR} \leftarrow J_{opt} - J_{wL}$
    else
      $J_{wR} \leftarrow CT^{++}(\mathcal{D}_R, d-1, \mathcal{UB})$
  $J_w \leftarrow J_{wL} + J_{wR}$
  if $J_{wL} = 0$ **then**
    $M_L \leftarrow w + 1$
  if $J_{wR} = 0$ **then**
    $M_R \leftarrow w - 1$
  if $J_w \leq J_{opt}$ **then**
    $\mathcal{UB} \leftarrow min(\mathcal{UB}, J_w), J_{opt} \leftarrow J_w$
    if $J_w = 0$ **then**
      break
  $Q.push(P_{NB}([i, j], u, J_w - \mathcal{UB}))$
  $\mathcal{V} = \mathcal{V} \cup \{w\}$
Output: J_{opt}

References

[1] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. *Proceedings of the AAAI Conference on Artificial*

Intelligence, 2020.

[2] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 2014.

[3] Kristin P. Bennett and Jennifer A. Blue. Optimal decision trees. *Pattern Recognition*, 1996.

[4] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 2017.

[5] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence, Vol.237*, 2016.

[6] Hendrik Blockeel, Laurens Devos, Benoît Frénay, Géraldin Nanfack, and Siegfried Nijssen. Decision trees: from efficient prediction to responsible ai. *Frontiers in Artificial Intelligence*, 2023.

[7] Matheus Guedes Vilas Boas, Haroldo Gambini Santos, Luiz Henrique de Campos Merschmann b, and Greet Vanden Berghe. Optimal decision trees for the algorithm selection problem: integer programming based approaches. *Internation Transactions in Operational Research*, 2021.

[8] Leo Breiman, Jerome Friedman, R.A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Taylor Francis, 1984.

[9] Catalin E. Brita, Jacobus G. M. van der Linden, and Emir Demirović. Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. *Association for the Advancement of Artificial Intelligence*, 2025.

[10] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2`, 2024.

[11] Emir Demirović, Emmanuel Hebrard Anna Lukina, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey. Murtree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 2022.

[12] Lucas Mentch Giles Hooker. Bridging breiman's brook: From algorithmic modeling to statistical learning. *Observational Studies, Volume 7, Issue 1*, 2021.

[13] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Elsevier Information Processing Letters*, 1976.

[14] Ana Kostovska, Diederick Vermetten Anja Jankovic, Sašo Džeroski, Tome Eftimov, and Carola Doerr. Comparing algorithm selection approaches on black-box optimization problems. *The Genetic and Evolutionary Computation Conference*, 2023.

[15] Rahul Mazumder, Xiang Meng, and Haoyue Wang. Quant-bnb: A scalable branch-and-bound method for optimal decision trees with continuous features. *International Conference on Machine Learning*, 2022.

[16] Fabian Pedregosa, Alexandre Gramfort Gaël Varoquaux, Vincent Michel, Bertrand Thirion, Olivier Grisel, Andreas Müller Mathieu Blondel, Joel Nothman, Peter Prettenhofer Gilles Louppe, Ron Weiss, Vincent Dubourg, Jake Vanderplas, David Cournapeau Alexandre Passos, Matthieu Brucher, Matthieu Perrot, and Ädouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 2011.

[17] Daniël Poolman. Optimal decision trees for the algorithm selection problem. Master's thesis, EEMCS, Delft University of Technology, 2024.

[18] John R. Rice. The algorithm selection problem. *Advances in Computers Vol. 15*, 1976.

[19] Cynthia Rudin. Modern decision tree optimization with generalized optimal sparse decision trees. Course Notes, 2023. Accessed: 2025-06-20.

[20] Giulio Segalini. Optimal decision trees for the algorithm selection problem a dynamic programming approach. Master's thesis, EEMCS, Delft University of Technology, The Netherlands, 2023.

[21] Ya Song, Laurens Bliek, and Yingqian Zhang. Revisit the algorithm selection problem for tsp with spatial information enhanced graph neural networks. *17th International Conference on Agents and Artificial Intelligence*, 2025.

[22] Jacobus G. M. van der Linden, Mathijs M. de Weerdt, and Emir DemiroviÄ. Optimal decision trees for separable objectives: Pushing the limits of dynamic programming. *arXiv:2305.19*, 2023.

[23] Jacobus GM van der Linden, Mathijs M de Weerdt, and Emir Demirović. Optimal decision trees for separable objectives: Pushing the limits of dynamic programming. *NeurIPS*, 2023.

[24] Sicco Verwer and Yingqian Zhang. Learning decision trees with flexible constraints and objectives using integer optimization. *Integration of AI and OR Techniques in Constraint Programming*, 2017.

[25] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, 2008.