Search Strategies for Optimal Decision Trees

Classification and Regression with Continuous Features

Mim van den Bos

Delft University of Technology

TU Delft

Search Strategies for Optimal Decision Trees

Classification and Regression with Continuous Features

by

Mim van den Bos

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Wednesday September 24, 2025 at 10:00 AM.

Student number:	5274559	
Project duration:	January 22, 2025 – September 24, 2025	
Thesis committee:	Dr. Emir Demirović,	TU Delft, supervisor
	Ir. Jacobus G. M. van der Linden,	TU Delft, daily supervisor
	Dr. Jérémie Decouchant,	TU Delft

Cover:	aerial photography of forest by Filip Zrnzević under Unsplash License
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `https://repository.tudelft.nl/`.

TU Delft

Preface

After submitting this thesis, I will be at the end of my five-year journey through higher education. During my time at the TU Delft and my minor in Singapore, I have learnt an incredible amount. In addition to the obvious technical knowledge, I have had incredible opportunities for growth. These are too numerous to list, but, among others, I have discovered a passion for teaching during my years as a teaching assistant, gained insight in a different culture by travelling to Singapore for my minor, and had the opportunity to present and publish my bachelor thesis at an international conference.

First and foremost I would like to thank my responsible supervisor Dr. Emir Demirović and my daily supervisor Koos van der Linden for their excellent guidance and for always being very engaged in our discussions.

Second, I thank my family and friends for their unconditional support while writing this thesis.

Third, I would like to thank everyone who has contributed to making my unique path through education possible.

Finally, I thank Thomas Maliappis for proofreading several sections of this thesis, to prevent emberrasing [sic] typos from making it into this final version of the thesis.

Mim van den Bos
Delft, September 2025

Abstract

Interpretable machine learning models, such as decision trees, are needed when decisions require trust. Optimal decision trees are shown to generalise better to new data than those constructed greedily, but due to the NP-hardness of the problem they are hard to apply to large datasets. Previous methods either do not take into account continuous features, are designed for a fixed maximum tree depth of two and three, or do not consider objectives other than classification.

We introduce *CODTree*, the first specialised branch-and-bound algorithm that finds optimal classification and regression trees with continuous features for an arbitrary maximum depth. Our algorithm is able to run different search strategies and includes a specialised solver for shallow trees. Our experiments show that global best-first search with a heuristic that prioritises smaller nodes with better lower bounds has the best time to optimality and anytime performance, and that the specialised solver for shallow trees provides a geometric mean speedup of 77.4x.

Compared to the state-of-the-art for classification, we have comparable runtime, but our search strategy has four orders of magnitude fewer operations for some datasets, although there are diminishing returns for greater depths. For regression, our algorithm is significantly faster than the state-of-the-art and, to the best of our knowledge, the first to find optimal regression trees at depth four.

Contents

1 Introduction

Small interpretable machine learning models, such as decision trees, regularly outperform their black-box counterparts on tabular data (Rudin 2019). Despite this, many high-stakes decisions are made using black-box models. This results in serious issues that can go undetected because the model is opaque, such as discrimination in criminal justice (Garrett and Rudin 2023) and healthcare (Amann et al. 2020). In contrast, transparency in the way a decision is made allows the user to (dis)trust the model to make high-stakes decisions (Du et al. 2019). Decision trees are intrinsically interpretable and non-linear, a combination that is shared by few other models (Barredo Arrieta et al. 2020). Therefore, techniques for finding decision trees need to be readily available for their broader adoption.

Traditionally, greedy heuristic methods, such as CART (Breiman et al. 1984), have been used to find decision trees. These methods use local statistics on the data to create branches in the tree, without considering the impact of future branches. This finds usable trees, but does not provide any guarantees to their performance. In extreme cases this leads to trees with exponentially more nodes than necessary (Garey and Graham 1974).

An *optimal decision tree* can be found by choosing the tree with the lowest possible regularised cost on the training data. Optimal decision trees are significantly smaller than greedy trees and generalise marginally better to out-of-sample data, making them a more interpretable and better model (Van der Linden et al. 2025). However, finding a tree with the lowest cost is NP-hard (Hyafil and Rivest 1976; Ordyniak and Szeider 2021), and thus difficult for large datasets with many possible tree configurations.

Recent works can find optimal trees for medium-sized datasets using specialised dynamic programming (DP) and branch-and-bound techniques (Aglin et al. 2020; Lin et al. 2020; Demirović et al. 2022). These works generally only consider a small fixed set of binary yes/no tests to divide the remaining data. There are, however, many possible tests, and selecting which tests are best to include is non-trivial. When a dataset has non-binary features, these methods choose the test set with a heuristic. For example, by binning data with continuous features (see Figure 1.1). This reduces the search space by limiting the number of tests to consider, but can negatively impact both accuracy and tree size (Mazumder et al. 2022; Briţa et al. 2025; Van der Linden et al. 2025).

In their list of the top ten challenges for interpretable machine learning, Rudin et al. (2022) claim that challenge #1 is optimising sparse decision models such as decision trees. They explicitly mention the lack of scalability and the lack of support for continuous features as two primary limitations. It is precisely these two limitations that this thesis addresses.

Supporting continuous features refers to considering all thresholds w of individual continuous features f of the form $f \leq w$. Including all these tests naively is prohibitively expensive, as it greatly increases the branching factor and exponentially increases the search space for methods that normally consider a small fixed set of tests. The set of threshold tests is not exhaustive, but is a common (Breiman et al. 1984) and reasonable selection that balances interpretability and discriminative power.

Some methods consider continuous features directly, but all have limited scalability. Mathematical

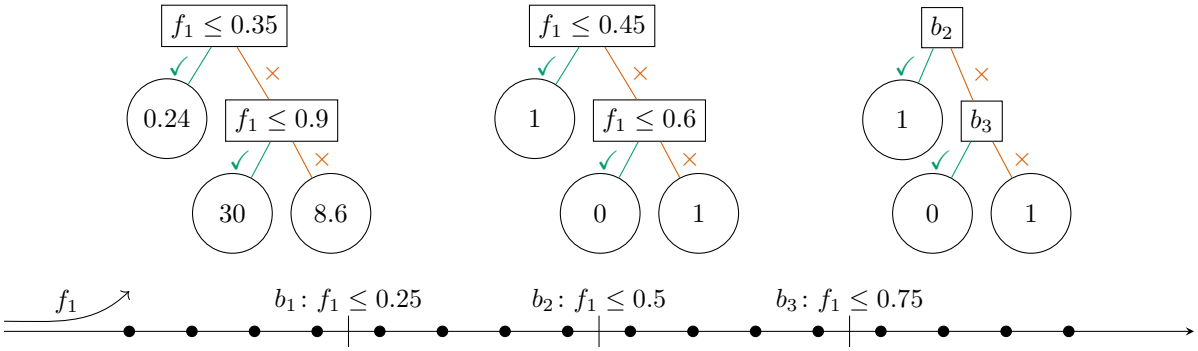


Figure 1.1: From left to right: a regression tree with continuous features, a classification tree with continuous features, and a classification tree with binned binary features. Black dots are instances at their feature value for f_1, between each point the data can be split. The vertical lines show an example of binning. Without a dataset, it is unclear if they are optimal.

formulations for generic solvers, such as mixed-integer programming, can often handle continuous features directly, but can take hours to solve datasets larger than a few thousand instances (Bertsimas and Dunn 2017; Shati et al. 2023). Some specialised algorithms take advantage of tests with the same feature and a similar threshold having similar results to prune large parts of the search space (Mazumder et al. 2022; Brița et al. 2025). Using these techniques, optimal *classification* trees can be found up to depth six for some datasets with a time limit of four hours (Brița et al. 2025). However, for optimal *regression* trees with continuous features, the current state-of-the-art finds shallow trees of depth two and three, the latter taking hours for many of their datasets (Mazumder et al. 2022).

This thesis focuses on increasing the scalability of finding optimal decision trees with continuous features, thereby making it viable for larger datasets. For feasibility, we limit the experimental scope of this thesis to classification and regression trees. We identify three major gaps to improve the runtime for finding optimal decision trees. We incorporate these into our new algorithm *CODTree*.[1]

First, the literature proposes various search strategies to improve scalability — depth-first (Aglin et al. 2020), best-first (Hu et al. 2019; Chaouki et al. 2025), and novel approaches (Mazumder et al. 2022; Demirović et al. 2023) — but their merit remains unclear. Comparisons are limited to those between entire methods, obscuring the impact of their search strategy. To the best of our knowledge, no optimal decision tree method directly compares different search strategies. Therefore, we design our algorithm to run with different search strategies. Our experiments show that global best-first search with a heuristic that prioritises smaller nodes with better lower bounds is significantly better than the others in terms of time to optimality and anytime performance. Compared to the state-of-the-art, we have comparable runtime, but our search strategy has four orders of magnitude fewer operations for some datasets, although with diminishing returns for greater depths.

Second, a specialised solver for trees of depth two has been shown to significantly speed up the search for binary features (Demirović et al. 2022), but it has not been attempted for continuous features. This solver is not directly suitable for continuous features, as it scales quadratically with the number of feature tests, but it is clear that the idea has merit. Following this idea, Brița et al. (2025) simultaneously solve the depth-one trees on the left and right sides of a subproblem. However, it is exclusively for classification and no attempt at a full depth two solver. Therefore, we adapt the latter to regression and propose pseudocode for a new specialised solver for trees of depth two. Our experiments show that the specialised solver for the left and right sides provides a geometric mean speedup of 77.4x.

Finally, previous work on optimal regression trees with binary features shows that an optimal solution to the k-means problem can be used as an effective lower bound (Zhang et al. 2023), but does not explore how this can be adapted to classification or continuous features. We show that a similar idea can be applied to classification and that, because of the many available splits with continuous features, this bound is sometimes an exact solution. Our experiments show no significant difference when this bound is used. We suggest a possible improvement and show that this improvement results in a tighter bound at the root level, but we cannot compute this improved bound efficiently.

[1] The reference implementation of CODTree is available at `https://github.com/mimvdb/codt`.

The remaining chapters are divided as follows. The next chapter provides an overview of related work. The third chapter covers the preliminaries necessary to understand the remainder of the work. The fourth chapter details our algorithm *CODTree* that we use to improve the scalability of optimal regression trees. In the fifth chapter, we experimentally show the difference in scalability between our and previous methods, clarify the previously unclear out-of-sample (OOS) advantage for optimal regression trees (Van den Bos et al. 2024), and the OOS advantage for directly accounting for continuous features in optimal regression trees specifically (previous results are conflated with classification (Mazumder et al. 2022)). In the final chapter, we draw our conclusions and provide suggestions for future work.

2

Related work

Heuristic approaches to finding classification and regression trees date back to the sixties, and a large body of research has developed since then. The following sections provide a brief overview of the previous literature on heuristic decision trees, semi-optimal decision trees, and optimal decision trees. Chapter 3 explains relevant concepts of these papers in more detail. For a more thorough review of decision trees, we refer the reader to Costa and Pedreira (2023).

2.1. Greedy classification and regression trees

Because finding optimal decision trees is NP-hard, early regression tree methods focus on finding greedy solutions with heuristics. AID was the first such regression tree heuristic (Morgan and Sonquist 1963). Recursively, it chooses the split with the least sum of square errors (SSE) and stops when such a split does not yield an improvement above a certain threshold. CHAID (Kass 1980; Biggs et al. 1991) extends AID with χ^2 significance testing, non-binary splits and classification. This method is used in the SPSS statistical software package.

GUIDE (Loh 2002) also uses χ^2 tests to prevent bias in variable selection toward variables with more distinct splits and fits linear models in the leaves. Variable selection bias is not eliminated by solving to optimality, but is not a focus of this study. We also do not compare to model extensions such as multiway splits and model trees, as it requires a careful evaluation of the difference in interpretability as well as out-of-sample performance.

For classification, ID3 (Quinlan 1986) and its extension C4.5 (Quinlan 1993) use information gain as the local heuristic. Unlike regression trees, where the objective is often used directly, they use information gain as a proxy for the accuracy objective, as this generally results in better subsequent divisions.

Perhaps the most widely used method in practice that supports both classification and regression trees is CART (Breiman et al. 1984). The main difference from AID is that CART grows an overly large tree. This large tree is then pruned to remove excess decisions by cross-validation. For classification, CART uses the Gini impurity as the local heuristic as opposed to the information gain in C4.5. An optimised version of CART is implemented in the popular scikit-learn Python package. At the time, they motivated the use of heuristics by claiming that optimal trees were not feasible.

2.1.1. Semi-optimal decision trees

Some heuristic methods include a limited search to provide a trade-off between scalability and the quality of the solution. These methods claim to find better trees than other heuristics, while not incurring the exponential cost of finding fully optimal trees. One such approach limits the search space by only considering the top-k features (ordered heuristically) for every split (Blanc et al. 2023). Another considers not just the best split for a depth-one tree, but the best split considering a depth-two tree (Kiossou et al. 2024).

Closely related are approaches that aim to find near-optimal trees. Stochastic gradient descent (Norouzi

et al. 2015) and local search (Dunn 2018) can be used to search for increasingly better trees. And finally, optimality can be sacrificed in places where it likely has a small impact on the overall accuracy (Babbar et al. 2025), or where a reference model expects no improvement (McTavish et al. 2022). However, none of these methods guarantee to find an optimal solution.

Semi-optimal methods empirically find trees that are close to optimal, but are not guaranteed to, and few show any bounds on the gap between the near-optimal and optimal solution. Garey and Graham (1974) show that greedy trees can have exponentially more nodes than the optimal solution. Blanc et al. (2023) show that there is a data distribution, parametrised by ϵ, where a decision tree with only the top-k features has an accuracy of at most $\frac{1}{2} + \epsilon$ while a tree with the top-$(k+1)$ features has an accuracy of at least $1 - \epsilon$.

In contrast, optimal methods are guaranteed to provide the optimal solution. Or, when the search is stopped early, provide an upper and lower bound on the solution. We discuss optimal methods that are designed to terminate the search early later in this chapter.

2.2. Optimal decision trees

We divide optimal decision tree methods into model-based and search-based approaches. Model-based approaches formulate a mathematical model of a decision tree and then use a general purpose solver to obtain solutions. These models can typically be easily extended to different objectives and features, but struggle to scale to large datasets.

Search-based approaches use a specialised solver to find optimal decision trees. These approaches scale to larger datasets by taking advantage of the structure of the problem. However, extending these approaches usually takes more effort. We provide a brief overview of both below.

2.2.1. Model-based approaches

Optimal decision trees regained attention when a mixed-integer programming (MIP) model showed that with advances in MIP solvers and computational power, optimal decision trees can be found for small datasets (Bertsimas and Dunn 2017).

Mathematical optimisation models are easily extendable, as is evidenced by the many variations available such as with hyperplane splits (Bertsimas and Dunn 2017), subset splits (Günlük et al. 2021), splits based on non-linear functions (Xue et al. 2024), linear models in the leaf nodes (Dunn 2018), randomised decisions at branching nodes (Blanquero et al. 2021), regression with absolute and squared error objectives (Bertsimas et al. 2017; Dunn 2018), and additional constraints such as fairness (Aghaei et al. 2019; Jo et al. 2023) and robustness (Justin et al. 2021; Vos and Verwer 2022).

There are also models that aim to find the smallest *perfect* tree, one that perfectly models the training data but has the least number of nodes. SAT is often used to model this problem (Narodytska et al. 2018; Janota and Morgado 2020; Shati et al. 2023). This thesis focuses only on minimising classification and regression loss.

The main downside of model-based approaches is that they do not scale to larger datasets. Several MIP (Verwer and Zhang 2019; Hua et al. 2022; Ales et al. 2024), MaxSAT (Shati et al. 2023), and constraint programming (CP) (Verhaeghe et al. 2020) models, all minimising classification loss, are focused on increasing scalability. However, they fail to scale to datasets larger than a few thousand instances. For a more extensive review of mathematical optimisation in decision trees, see Carrizosa et al. (2021).

2.2.2. Specialised search-based approaches

In contrast to model-based approaches, specialised search-based approaches use algorithms specifically tailored to optimise decision trees. Such algorithms have existed since the sixties (Reinwald and Soland 1966) for finding perfect trees of minimum size or test cost. For a review of the methods at that time, see Moret (1982). In the remainder of this chapter, we focus on the more recent literature after the resurgence of interest after the work of Bertsimas and Dunn (2017).

Branch-and-bound and dynamic programming Hu et al. (2019) adapt an optimal decision list algorithm to decision trees. They introduce a specialised algorithm that finds optimal decision trees faster than any model-based method before it using branch-and-bound search to prune parts of the search space that are proven not to contain the optimal solution.

Aglin et al. (2020) use ideas from itemset mining to construct optimal decision trees. They extend their earlier method (Nijssen and Fromont 2007) with branch-and-bound search and use dynamic programming (DP) to reuse solutions from other subtrees that use the same set of feature tests.

Lin et al. (2020) show that specialised solvers can be generalised to multiple objectives. They are the first to show progress in finding optimal decision trees with the full range of thresholds of continuous features. By finding bounds for trees with similar features, they can find optimal trees with feature tests for all possible thresholds for datasets with a hundred thresholds. However, their experiments show that the slowdown gets much larger as more thresholds are included. They use smaller sets of tests for their other experiments, presumably because this allows them to find deeper trees.

Depth-two solver and similarity bound Demirović et al. (2022) introduces a new algorithm that uses a specialised solver for trees of depth two and proposes a bound for similar datasets. However, their specialised solver for trees of depth two requires quadratic memory for the number of feature tests, which is prohibitive for the full range of continuous features. In addition, their bound for similar datasets works well with depth-first search order, since trees traversed subsequently differ by a single feature test, while we investigate many different search orderings.

Van der Linden et al. (2023) show that many optimisation techniques do not rely on the specific objective function of an optimal decision tree and can be applied more generally. However, they also do not consider the full range of continuous feature tests.

Van den Bos et al. (2024) adapt the specialised solver for trees of depth two from Demirović et al. (2022) to regression by decomposing the per-instance cost into three running sums from which the SSE can be derived. They additionally optimise trees with (simple) linear regression models in the leaf nodes.

Cache Memory can be a limiting factor when combining very large datasets with dynamic programming in an environment with memory constraints. Aglin et al. (2023) limit the unbounded growth of caches with an eviction policy that keeps the entries most relevant to the search.

Anytime performance For datasets where an optimal solution is infeasible to compute, terminating the algorithm early might still yield a good tree. Several changes to the search order are proposed to improve this *anytime performance* of the algorithm. Demirović et al. (2023) interleave the solving of left and right subtrees, to avoid spending all the initial time on one side of the tree. Kiossou et al. (2022) first consider trees that are more similar to a tree chosen heuristically. Finally, Chaouki et al. (2025) use AND/OR graph search to first explore the most promising solution and find the optimal solution in fewer steps. However, these all assume a small set of feature tests, without taking into account the large amount of similar tests for continuous features.

k-Means lower bound Branch-and-bound prunes the search space by bounding the error. Lower bounds are usually found by exhaustively searching a part of the search space. Zhang et al. (2023) propose a lower bound for regression that can be computed immediately without search. The search problem can be relaxed by assuming that the data can be partitioned in any way by the feature tests. In a regression setting, this relaxation is equivalent to solving the univariate k-means problem where k is the number of leaves. They extend a linear DP algorithm to solve the univariate k-means (Wang and Song 2011; Song and Zhong 2020) with an early-exit condition if increasing k improves the solution less than a regularisation term. Furthermore, they augment this bound with the guaranteed error of instances that match exactly in features, but differ in label.

Continuous features Mazumder et al. (2022) find complete optimal classification and regression trees up to depth three with all possible continuous feature tests, even for datasets with tens of thousands of instances. They manage the large search space by recursively investigating solutions at

quantiles of the remaining thresholds and then pruning entire ranges of thresholds at once. However, finding trees of depth three can take hours, and their method does not generalise to higher depth.

Brița et al. (2025) find optimal decision trees of arbitrary depth with all possible continuous feature tests. They prune ranges of thresholds around a solution by considering the minimum difference that is needed for an optimal solution and solve shallow trees exhaustively. The search order they use is depth-first for choosing features, recursively splitting the remaining thresholds in half until the range can be pruned.

Staus et al. (2025) also consider all possible thresholds for continuous features, but only search for perfect trees and minimise the number of branching nodes. We minimise loss without assuming a perfect tree exists.

Gap We repeat the three gaps in the literature that this thesis addresses. First, only Chaouki et al. (2025) explicitly focus on the search strategy with AND/OR search, and no work compares the isolated impact of the search strategy on scalability. Our algorithm is able to use different search strategies. Second, a specialised depth-two solver improves scalability for a small number of feature tests (Demirović et al. 2022), but only a solver that solves the left and right depth-one trees simultaneously has been attempted for continuous features (Brița et al. 2025). We adapt the left and right solver to regression and propose pseudocode for a full depth-two solver. Third, the k-means lower bound is effective for regression, but is not used for classification, nor does it take advantage of the many feature tests available with continuous features. We adapt this bound to classification and propose two improvements to this bound. One improvement is a fast check to see if the bound is exact, the other looks ahead in the search to refine the bound.

3

Preliminaries

This chapter first introduces the common terminology and notation used in the remainder of the work. This is followed by an overview of search strategies for optimal decision trees. Lastly, we show existing algorithmic techniques to reduce the search space. Specifically, we show bounds on the error, specialised solvers for shallow trees, caching, and preprocessing. For convenience, Table 3.1 summarises the latter for current implementations in the literature.

3.1. Notation

A *decision tree* is an *axis-aligned* partition of the feature space, where each part is assigned a single prediction. In a decision tree τ, each instance takes a path through the *branching nodes* of the tree, and a decision is made based on the *leaf node* reached. A branching node performs a binary threshold test on a single feature of the instance. A leaf node has a constant label irrespective of the instance.

A decision tree for data with features $\mathcal{F} = \{f_1, f_2, \dots\}$ is a function $\tau : \mathbb{R}^{|\mathcal{F}|} \to \mathcal{Y}$ where the output is a prediction and the input a feature vector. We consider decision trees that solve classification and regression problems. For *classification trees*, $\mathcal{Y} = [0, C-1]$ with C the number of classes and for *regression trees*, $\mathcal{Y} = \mathbb{R}$. We say that the size of the tree $|\tau|$ is equal to the number of branching nodes in the tree.

An *optimal* decision tree is one that minimises a loss function $\mathcal{L}$ for a maximum depth d. We include a subscript c or r when a function is applicable only to classification or regression, respectively. For classification, the loss is the number of misclassifications (Equation (3.1)), and for regression the sum of squared errors (SSE, Equation (3.2)), which is equivalent to optimising the mean squared error (MSE). Note that we can easily compute the label of a leaf node that minimises $\mathcal{L}$: for classification the class with the most instances (Equation (3.4)) and for regression the mean of all labels (Equation (3.5)).

To control the size of the tree, we add a regularisation parameter λ that penalises a larger tree. Although an explicit node budget can control the tree size more directly, it significantly increases the size of the search space at an equal maximum depth and does not result in better out-of-sample performance (Van der Linden et al. 2025).

Table 3.1: Overview of specialised optimal decision tree methods for (C)lassification and (R)egression.

Method	Tree	Traversal	Similarity bound	D2 solver	Caching	Reference
Continuous features						
CODTree	C/R	Many	Threshold	Yes	No	This thesis
ConTree	C	DFS	Threshold	Yes	Dataset	Brița et al. (2025)
Quant-BnB	C/R	Unique	Threshold	No	No	Mazumder et al. (2022)
Binary features						
DL8.5	C	DFS	No	No	Branch	Aglin et al. (2020)
MurTree	C	DFS	Direct	Yes	Dataset	Demirović et al. (2022)
STreeD	C/R	DFS	Direct	Yes	Dataset	Van der Linden et al. (2023) and Van den Bos et al. (2024)
OSDT	C	BFS	No	No	Branch	Hu et al. (2019)
GOSDT	C	BFS	Feature	No	Dataset	Lin et al. (2020)
OSRT	R	BFS	No	No	Dataset	Zhang et al. (2023)
Blossom	C	Unique	No	No	No	Demirović et al. (2023)
LDS-DL8.5	C	Unique	No	No	Branch	Kiossou et al. (2022)
Branches	C	AO*	No	No	Branch	Chaouki et al. (2025)

$$e_c(y, \hat{y}) = \mathbb{1}(y \neq \hat{y}) \tag{3.1}$$

$$e_r(y, \hat{y}) = (y - \hat{y})^2 \tag{3.2}$$

$$\mathcal{L}(\tau, \mathcal{D}) = \lambda|\tau| + \sum_{(x,y)\in\mathcal{D}} e(y, \tau(x)) \tag{3.3}$$

$$\mathcal{L}abel_c(\mathcal{D}) = \operatorname*{argmax}_{c\in\mathcal{Y}} \sum_{(x,y)\in\mathcal{D}} \mathbb{1}(y = c) \tag{3.4}$$

$$\mathcal{L}abel_r(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} y \tag{3.5}$$

A dataset $\mathcal{D}$ is a set of instances $(x, y) \in \mathcal{D}$ with a feature vector $x \in \mathbb{R}^{|\mathcal{F}|}$ and a label $y \in \mathcal{Y}$. Let x^{f_i} indicate the value of feature f_i for x. We define a shorthand for selecting a subset of a dataset as $\mathcal{D}(f_i \leq w) \triangleq \{(x, y) \in \mathcal{D} \mid x^{f_i} \leq w\}$.

We find the optimal threshold tests of the form $f_i \leq w$ for all real w. Note that we do not need to consider the entire continuous range as a slight shift will not affect how the feature test partitions the data and, therefore, will not affect the objective. For this reason, we only consider the midway point between consecutive unique feature values.

We denote the set of thresholds for feature f_i as W^{f_i}, and refer to the jth smallest threshold as $w_j^{f_i}$. When the feature is clear from the context, we leave out the feature in the superscript. We only directly consider threshold tests for continuous features, but note that ordinal and categorical features can be encoded as one or more continuous features.

Now we can recursively formulate the loss of the optimal decision tree OPT that minimises $\mathcal{L}$ as follows.

$$l(\mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} e(y, \mathcal{L}abel(\mathcal{D})) \tag{3.6}$$

$$OPT(\mathcal{D}, d) = \begin{cases} l(\mathcal{D}) & \text{if } d = 0 \vee l(\mathcal{D}) \leq \lambda \\ \min \begin{cases} l(\mathcal{D}) \\ \min\limits_{f\in\mathcal{F}, w\in W^f} \lambda + OPT(\mathcal{D}(f \leq w), d-1) \\ \qquad\qquad + OPT(\mathcal{D}(f > w), d-1) \end{cases} & \text{otherwise} \end{cases} \tag{3.7}$$

3.2. Searching for a tree

The literature uses a variety of search strategies, such as depth-first and best-first. We briefly introduce the optimal decision tree problem as a graph search, and discuss the benefits and drawbacks of the search strategies introduced in the literature.

Figure 3.1 illustrates a partial search graph to find an optimal decision tree. The initial partial search graph consists of the root node, with each child representing a possible feature test in that node. The search graph can be further expanded by adding a left and right child, with their own feature test children, to a feature test. Repeated expansion until the depth limit is reached constructs the full search graph.

Decision trees are represented in the search graph as connected subgraphs that contain the root, contain at most one feature test for each node, and contain the left and right nodes for each selected feature test. The optimal decision tree can be found by considering all such subgraphs.

In practice, branch-and-bound search is used to prune large parts of the search graph without constructing it fully. Each decision tree gives an upper bound on the possible error. We discuss lower bounds in Section 3.3 and Section 3.4. Bounds propagate upwards; for example, the upper bound for a feature test is equal to the sum of its left and right nodes. When the lower bound for a node or feature test is greater than the upper bound, it can be pruned.

How the search graph is traversed impacts the rate at which optimality is proven, the quality of the solution after a short period of search (anytime performance), and the complexity and memory cost of the algorithm. However, all search strategies have an exponential worst-case runtime, except if P = NP. Hence, we focus on the average-case runtime. Several traversals have been attempted in the literature, which we summarise below.

3.2.1. Depth-first search

Depth-first search (DFS) is perhaps the simplest way to traverse the search tree. For a fixed ordering of the features and thresholds, pick the first and keep expanding the last added node to exhaustion. Once it has been exhaustively searched, pick the next until all options have been exhausted and the entire search space is traversed. It is used in DL8.5 and its derivatives, such as MurTree and STreeD. Note that while the order needs to be fixed, a heuristic may be used for this fixed ordering.

The advantages of using DFS are that solving the left subtrees fully first, means the solution can be used for tighter bounds on the error of the right tree. The high locality of the search also ensures similar trees are explored close together, this can help for bounds that depend on similar subtrees such as the direct use of the similarity lower bound described further in Section 3.3. The memory cost of the search tree is at most the size of a single full tree, since the previously iterated trees can be discarded, and future iterated trees have no state. The cache of previously computed solutions is therefore the limiting factor for memory.

There are two main disadvantages of using DFS. The first is that the DFS search order cannot explore heuristically more interesting parts of the search graph until the current subproblem is finished, so the information gained during search is not fully exploited. The second is that before the left subproblem is solved, the right subproblem is never considered. This negatively affects the anytime performance because only the left part of the tree has been searched. For any dataset too large to be solved to optimality, the right subtree is likely still very suboptimal.

3.2.2. Balanced search

Because the most notable drawback of DFS is its limited anytime performance, some methods are dedicated to improve this. Blossom (Demirović et al. 2023) completely expands a tree before moving on to alternatives, while LDS-DL8.5 (Kiossou et al. 2022) uses limited discrepancy search to first search for solutions with only a small deviation from a heuristic solution.

The advantage of balanced methods is that they are able to get higher accuracy than greedy on datasets where fully solving to optimality is not an option. The disadvantage is that these methods are specifically designed for anytime performance and do not improve, and sometimes worsen, the time to prove optimality. For example, Blossom lacks a cache that is more useful when fully exploring the search space, and LDS-DL8.5 requires restarting the search multiple times.

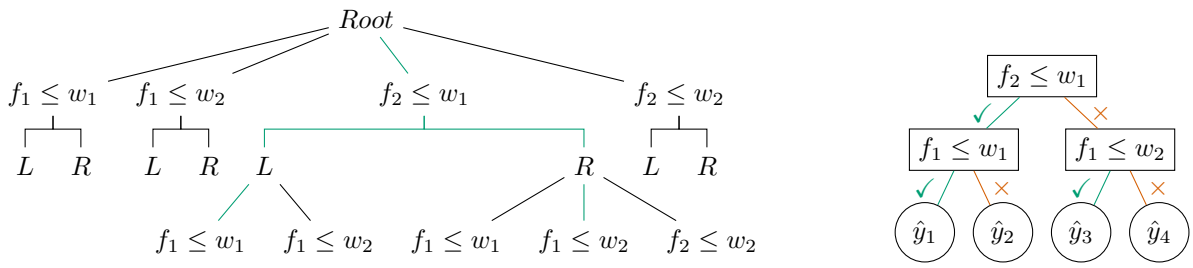


Figure 3.1: On the left, a partially expanded search graph for a depth two tree with features f_1 and f_2 each with two thresholds w. On the right, the tree corresponding to the highlighted solution subgraph with label predictions $\hat{y}$. Forked edges indicate that feature tests are AND nodes, its left and right child both need to be solved.

3.2.3. Global best-first search

To explore the most promising search nodes first, some methods resort to best-first search. They attach a heuristic priority value to all search nodes and repeatedly expand the search node with the highest priority (lowest value) one step. The main advantage of best-first search is that the search can dynamically decide to try a different node based on information gained during the search. The main disadvantage is that memory usage increases when many search paths are partially explored and the solution has not yet been found. We now discuss the different priority heuristics used in the literature.

Lower bound Hu et al. (2019) use two different heuristics for node priority, the first is to prioritise the node with the lowest current lower bound on the loss first. After all, if the lower bound remains low after further search, then this is the solution with the lowest error.

Curiosity The second lower bound introduced by Hu et al. (2019) is curiosity. Translated into our setting, curiosity is the current lower bound divided by the size of the dataset that remains in that node. This gives lower priority to nodes that represent a smaller subset of the total problem, such as very deep nodes or unbalanced splits. They observe a reduction in runtime by a factor of two and memory by a factor of four using curiosity instead of the lower bound.

GOSDT GOSDT (Lin et al. 2020), also uses a heuristic based on the lower bound and the size of the subproblem: the lower bound minus the size of the dataset remaining in that node (in their highest first setting, size of the dataset minus the lower bound).[1] In addition, they always prioritise propagation of updated bounds up the tree over exploring new nodes. Similarly to curiosity, this reduces the priority of nodes that represent a smaller subset of the problem. By extension, Zhang et al. (2023) use the same heuristic for regression.

Although these heuristics use similar parameters, the chosen heuristic has a large impact on runtime. We discuss this further in Section 5.3.

3.2.4. AND/OR best-first search

Chaouki et al. (2025) argue that the use of a single global queue is not sufficient, and that the problem should instead be approached as an AND/OR graph search. An AND/OR graph has AND nodes that require all children to be solved, while an OR node requires only a single child to be solved. In the context of the search graph, the root, left, and right nodes are OR nodes, as a single feature test should be selected. The feature tests are AND nodes, as they require both the left and right sides to be solved. Figure 3.1 highlights a solution subgraph in the AND/OR search graph.

Approaching decision tree optimisation as an AND/OR graph stems from a very early branch-and-bound method (Martelli and Montanari 1978), later generalised and named AO* (Nilsson 1980), and four decades later applied again to the problem of optimal decision trees (Verhaeghe et al. 2020; Sullivan et al. 2024; Chaouki et al. 2025).

[1] This priority heuristic is not discussed in their paper, but is included in the implementation of GOSDT, `https://github.com/ubc-systopia/gosdt-guesses`, b9116d0557613dc42ff1fcefcf0bc0be91db1620

AND/OR search always selects the subproblem that currently provides the lower bound on the error of the entire problem by hierarchically descending through the search graph. This entirely circumvents the issue that the curiosity and GOSDT heuristics for global best-first search attempt to mitigate. In a global queue, where the next problem to be solved can be a small subproblem of an ancestor node with a high lower bound, the size of the remaining data indicates the importance of solving it. All of the heuristics for global best-first search are identical in AND/OR best-first search, as the size of the subproblem is constant in each subproblem.

3.3. Similarity lower bounds

A primary technique for bounding subproblems is by comparing them to previously found solutions to similar subproblems. The similarity lower bound considers two datasets $\mathcal{D}$ and $\mathcal{D}'$ and computes a lower bound $LB_{\mathcal{D}} \leq OPT(\mathcal{D}, d)$, given a solution $OPT(\mathcal{D}', d)$. In this section, we prove this lower bound in a way similar to Demirović et al. (2022).

We take two intermediate steps to arrive at the bound. First, we define the function $e^{max}(\mathcal{D}, E)$ as the worst contribution the instances $E \subseteq \mathcal{D}$ could have to the optimal loss for a dataset containing at most the instances in $\mathcal{D}$. For classification, this is Equation (3.8) because each instance can be misclassified. For regression, we obtain the bound Equation (3.9) by considering the extreme points of the dataset (Van den Bos et al. 2024), because a leaf node label is never less than the minimum label in the dataset, nor greater than the maximum label in the dataset (Dunn 2018).

$$e_c^{max}(\mathcal{D}, E) = |E| \tag{3.8}$$

$$e_r^{max}(\mathcal{D}, E) = \sum_{(x,y)\in E} \max(y - y_{min}(\mathcal{D}), y_{max}(\mathcal{D}) - y)^2 \tag{3.9}$$

$$y_{min}(\mathcal{D}) = \min_{(x,y)\in\mathcal{D}} y \tag{3.10}$$

$$y_{max}(\mathcal{D}) = \max_{(x,y)\in\mathcal{D}} y \tag{3.11}$$

Second, we prove two special cases that we use as lemmas in the final proof. The first is where $\mathcal{D}' \subseteq \mathcal{D}$, Lin et al. (2020) refers to this as the subset bound. The second is where $\mathcal{D} \subseteq \mathcal{D}'$, we will refer to this as the superset bound. The remainder of this proof loosely follows the proof in Demirović et al. (2022), generalised to include regression as well as classification.

Lemma 3.1. For any $\mathcal{D}' \subseteq \mathcal{D}$ it holds that $OPT(\mathcal{D}', d) \leq OPT(\mathcal{D}, d)$.

Proof. Let $E = \mathcal{D} \setminus \mathcal{D}'$. Since $OPT(\mathcal{D}', d)$ is optimal and the contribution of any $(x, y) \in E$ to the loss cannot be negative, we have $OPT(\mathcal{D}', d) \leq OPT(\mathcal{D}' \cup E, d) = OPT(\mathcal{D}, d)$. □

Lemma 3.2. For any $\mathcal{D} \subseteq \mathcal{D}'$ it holds that $OPT(\mathcal{D}', d) - e^{max}(\mathcal{D}', \mathcal{D}' \setminus \mathcal{D}) \leq OPT(\mathcal{D}, d)$.

Proof. Let $E = \mathcal{D}' \setminus \mathcal{D}$. Since $OPT(\mathcal{D}', d)$ is optimal and by the definition of f, we have $OPT(\mathcal{D}', d) - e^{max}(\mathcal{D}', E) \leq OPT(\mathcal{D}' \setminus E, d) = OPT(\mathcal{D}, d)$. □

Theorem 3.3. For two arbitrary datasets $\mathcal{D}, \mathcal{D}'$, it holds that $OPT(\mathcal{D}', d) - e^{max}(\mathcal{D}', \mathcal{D}' \setminus \mathcal{D}) \leq OPT(\mathcal{D}, d)$.

Proof.

$$\begin{aligned}
LB_{\mathcal{D}} &= OPT(\mathcal{D}', d) - e^{max}(\mathcal{D}', \mathcal{D}' \setminus \mathcal{D}) && && (3.12)\\
&= OPT((\mathcal{D} \cap \mathcal{D}') \cup (\mathcal{D}' \setminus \mathcal{D}), d) - e^{max}(\mathcal{D}', \mathcal{D}' \setminus \mathcal{D}) && \text{rewrite} && (3.13)\\
&\leq OPT(\mathcal{D} \cap \mathcal{D}', d) && \text{by Lemma 3.2} && (3.14)\\
&\leq OPT(\mathcal{D}, d) && \text{by Lemma 3.1} && (3.15)
\end{aligned}$$

□

The remainder of this section describes how this idea has been applied in previous works.

Feature substitution Lin et al. (2020) give a bound on the error of a tree when changing the feature in the root node. In their context, where each feature f_i is binary, they assume neighbouring features are likely binarisations of the same continuous feature and therefore similar. For each subproblem, they keep track of the lower and upper bounds for trees with root feature f_i, and update the bounds for f_{i+1} and f_{i-1} according to the similarity bound.

Direct comparison Demirović et al. (2022), Van der Linden et al. (2023), and Van den Bos et al. (2024) directly compare each subproblem with two recently used datasets for each depth d. They use the same assumption that subsequent binary features f_i are similar. They use two distinct datasets because, while the left and right subproblems partition the data and are therefore not similar, subsequent left subproblems and subsequent right subproblems are likely similar.

The key difference from feature substitution lies in their depth-first traversal. Due to the traversal, the most similar recently used dataset at the same depth is likely f_{i-1}. However, also due to the traversal, the bound for this previous dataset is already exact, while f_{i+1} is not used as no bound has been computed yet.

In addition, they use similarity to incrementally recompute the error incurred after branching on any pair of binary features. These pairwise costs are used in a procedure to quickly compute the optimal tree with a maximum depth of two.

Threshold pruning After choosing the threshold w_1 for a feature f_i to split on, there are two resulting subproblems. One for the left branch, where $f_i \leq w_1$, and one for the right branch where $f_i > w_1$. Lin et al. (2020) make the observation that for a different threshold w_2, the subproblems for the first can act as a lower bound for the subproblems of the second. If $w_1 \leq w_2$, then the instances in the left branch of $f_i \leq w_1$ are a subset of the instances in the left branch of $f_i \leq w_2$ and the subset bound applies (Lemma 3.1). Vice versa, if $w_1 \geq w_2$, then the right branch of $f_i \leq w_1$ is a subset of the instances in the right branch of $f_i \leq w_2$.

Mazumder et al. (2022) and Briță et al. (2025) apply this to prune entire subsets of thresholds $[w_i, w_j]$ at a time, by setting the lower bound to the left branch of a solution with threshold less than or equal to w_i and the right branch of a solution with threshold greater than or equal to w_j.

Mazumder et al. (2022) make the bound more specific by adding the penalty of an optimal tree trained only on the instances landing between thresholds $(w_i, w_j]$. This is an underestimate of the true error contribution for these instances and can therefore be added to the bound.

Briță et al. (2025) also apply the similarity lower bound after finding a solution by counting the minimum number of instances that need to change sides, and prune that amount of thresholds around the solution.

3.4. k-Means equivalent points lower bound

For optimal regression trees, a useful relaxation is to ignore the feature values. If we only consider $k = 2^d$ possible leaves, the optimal way to divide the remaining instance labels is to create k clusters (leaves) of instance labels such that the sum of squared distances to the mean of their respective cluster is minimised. This problem is referred to as the k-means problem and is a lower bound for the error. Note that it is a relaxation of the problem, since the features of the instances may not allow the instance labels to be divided optimally. Zhang et al. (2023) show the effectiveness of this bound for optimal regression trees with binary features.

This bound relies on an efficient solution to the one-dimensional k-means problem. Bellman (1973) gives the DP formulation for this problem. We rephrase it for our context as follows. For ease of notation, we introduce a new feature f_y whose feature values are equal to the label y of each instance. Consider clusters as a sequence of intervals $(0, i_1], (i_1, i_2], ..., (i_{k-1}, |W^{f_y}|]$, where $i_j < i_{j+1}$ and $w_{i_j} \in W^{f_y}$, the k-clustering with minimal sum of squared errors (SSE) is now

$$S(k, i) = \min_{k \leq j \leq i} (l(\mathcal{D}(f_y \geq w_j)(f_y \leq w_i)) + S(k-1, j-1)) \tag{3.16}$$

The natural DP solution is to fill a two-dimensional DP matrix based on this formulation. There are $\mathcal{O}(k|W^{f_y}|)$ cells in this matrix, each computing $\mathcal{O}(|W^{f_y}|)$ options for i. Wang and Song (2011) note that the SSE does not need to be recomputed each time, so the total time complexity is $O(k|W^{f_y}|^2)$ (see also Appendix A). However, Song and Zhong (2020) note that the time complexity can be improved to $O(k|W^{f_y}|)$ by introducing a *totally monotone* matrix A.

$$A(k)_{i,j} = \begin{cases} S(k-1, j-1) + l(\mathcal{D}(f_y \geq w_j)(f_y \leq w_i)) & \text{if } 1 \leq k \leq j \leq i < |W^{f_y}| \\ \infty & \text{otherwise} \end{cases} \tag{3.17}$$

The optimal clustering can now be found by finding the row minima of A. Let $j(k, i)$ be the j selected to minimise $S(k, i)$, so the minimum column of row i in $A(k)$. The matrix $A(k)$ is monotone because $i_1 < i_2 \implies j(k, i_1) \leq j(k, i_2)$, and totally monotone because all submatrices are also monotone. The SMAWK algorithm takes advantage of this property to find the minima of the rows with $\mathcal{O}(|W^{f_y}|)$ probes in the A matrix without fully instantiating it (Aggarwal et al. 1987). The main idea behind the SMAWK algorithm is that because of total monotonicity, each probe can remove either a row or a column from the matrix.

Zhang et al. (2023) make two additional refinements when applying the k-means to optimal regression trees. First, they incrementally compute the cluster matrices from one to k, and stop early if the next iteration cannot improve the solution more than the cost of adding another branch λ.

Second, they note that instances with identical feature vectors can never be assigned to different leaves in any decision tree, resulting in a minimum error for these instances. They call this the equivalent points bound and implement this by merging instances with identical feature vectors to a single instance with a mean label and a weight of the number of merged instances. The weighted one-dimensional k-means can be computed similarly, so they call the full bound the k-means equivalent points lower bound.

3.5. Depth-two solver

MurTree (Demirović et al. 2022) introduces a specialised solver for classification trees of depth two. The lowest level of the search tree is where most of the work happens, so accelerating these specifically achieves a large overall speed-up.

For each pair of feature tests, they count the number of instances in each class that satisfy both tests. Subtracting those from the total derives the negations of those tests. In practice, this is faster than counting the error for each split separately.

This also takes advantage of sparsity in the feature tests, as each instance only adds to the count of a feature test pair when it satisfies both feature tests. Note that each feature test is satisfied for at most half of the instances; when it is satisfied for more, it can be negated during pre- and post-processing.

The depth-two solver can be extended to other objectives in some cases (Van der Linden et al. 2023). In particular, it can be used for regression (Van den Bos et al. 2024).

Since it needs a count for each pair of feature tests, the memory cost is on the order of the number of tests squared. For their setting, binary features, this works fine. However, for continuous features, the number of possible tests is much larger, and the memory cost becomes prohibitively expensive. For example, for the *bean* dataset, there are 211966 feature tests. Even if we assume a single eight-byte integer is enough to store the counts, this would require approximately 335GB of memory. Mazumder et al. (2022) therefore report an out-of-memory error for four out of the sixteen datasets they tested.

ConTree (Briță et al. 2025) offers a similar specialised depth-two solver suitable for continuous features, but for a fixed test in the root node. The usual search is used for the test in the root node. It solves the left and right depth-one subtrees simultaneously by implicitly splitting the data. This avoids the cost of explicitly splitting the data. Their solver also does not account for non-complete trees, as their objective does not have a sparsity penalty. Furthermore, they assume a finite number of labels, which is not directly applicable to regression.

3.6. Caching

Solutions to subproblems can be stored for later use to save computation. Using the dataset as the key allows the most reuse for solutions, but can require a large memory for large datasets. Alternatively, the features and thresholds used can be used as a cache key. This allows less reuse, but requires less memory. Demirović et al. (2022) compare branch caching and dataset caching for binary feature tests.

The deeper the tree, the more subproblems that need to be cached. Aglin et al. (2023) recognise this problem and suggest a bounded cache. Their approach clears a part of the cache when it is filled and prioritises keeping the parts of the cache that are likely still useful. Bounding the cache creates a trade-off between runtime and memory usage, since the cleared subproblems need to be solved again when encountered, but allows deeper trees to be found with less memory. A bounded cache may be necessary when the number of sub-problems grows.

Although caching is effective for small sets of feature tests, our preliminary experiments show almost no repeated subproblems using continuous features. In addition, some search strategies are more demanding of memory than others, and adding a cache makes the cause of an out-of-memory error less clear. For this reason, we do not implement a cache.

4

Method

Our main contribution is *CODTree*, a branch-and-bound algorithm to find optimal classification and regression trees with an arbitrary maximum depth. The algorithm iteratively builds the search graph described in the preliminaries (see Section 3.2). The main difference from other algorithms is that this algorithm supports multiple search strategies. This allows us to experimentally assess the impact of the search strategy. We incorporate and adapt several of the ideas presented in the preliminaries and focus on selecting an effective search strategy.

In this chapter, we present and discuss our algorithm. We discuss it in the following order.

1. We cover the main steps of our algorithm.
2. We discuss what an ideal search strategy should accomplish, what strategies we will consider in our experiments, and formally define each.
3. We elaborate on the details of the algorithm and how we prune solutions.
4. We cover several alternatives to exhaustively search for shallow optimal trees, which in practice increases the scalability because it accelerates the search in the part where most time is spent.
5. We extend the idea of the k-means lower bound to a general clustering lower bound and show that this lower bound is exact in some cases.

4.1. Main algorithm

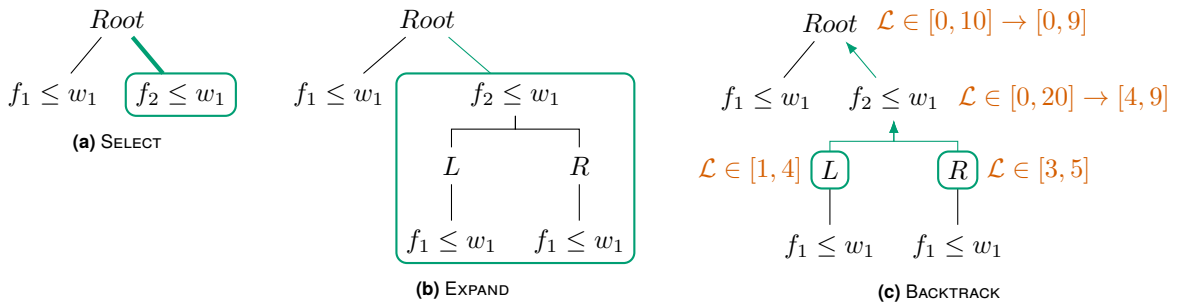
Our search algorithm consists of three core operations: SELECT, EXPAND, and BACKTRACK. We illustrate these in Figure 4.1. It is similar to other AND/OR graph search algorithms (Martelli and Montanari 1978; Nilsson 1980; Chaouki et al. 2025). We give a brief overview of Algorithm 1 below, and elaborate on the details of the subroutines in Section 4.3. Our search graph is a tree, but we avoid calling it such to avoid confusion with decision trees.

Initially, we construct the search graph as the root node with a queue of remaining feature tests. Then, we repeat the three core operations until we have found an optimal tree.

First, we SELECT the path from a leaf in the search graph to explore further. We say that a *path* consists of a sequence of tuples (N, f_k, w), where N is a node with selected feature test $f_k \leq w$. The last node in a path is always the root. The path in Figure 4.1a has a single step: $(Root, f_2, w_1)$.

Second, we EXPAND the selected leaf. We do this by splitting the remaining data at the selected node by the selected feature test. These split datasets are then used to initialise the left and right nodes and provide a more refined bound on the error. In Figure 4.1b, the initialised left and right nodes have a single possible feature test remaining: $f_1 \leq w_1$.

Finally, we BACKTRACK through the search graph to update the upper and lower bounds of each node and feature test. We walk the selected path back from leaf to root, pruning any feature test in which the upper bound is now strictly lower than the lower bound. In Figure 4.1b, we find a lower bound of one



(a) SELECT **(b)** EXPAND **(c)** BACKTRACK

Figure 4.1: Search graph during a single iteration of the algorithm. **(a)** selecting a feature test based on the search strategy, **(b)** expanding the left and right node of the feature test, **(c)** backtracking the bounds on the loss $\mathcal{L}$. Repeating these three core operations until the search graph is fully explored results in the optimal tree.

and three for the left and right nodes, respectively, and an upper bound of four and five, respectively. These are summed to set the new lower and upper bounds of the feature test to four and nine. Finally, the root can use the upper bound of the feature test, but not the lower bound, as a different feature test might have a lower loss.

Algorithm 1 SEARCH$(\mathcal{D}, d)$ finds the optimal loss of a decision tree with maximum depth d for the dataset $\mathcal{D}$. For clarity, the given algorithm returns the loss, our implementation returns the tree.

$Root \leftarrow$ CONSTRUCTNODE$(\mathcal{D}, d)$
while $LB(Root) < UB(Root)$ **do**
  $Path \leftarrow$ SELECT$(Root)$
  $(N, f_k, w) \leftarrow Head(Path)$
  EXPAND(N, f_k, w)
  while $Path$ *is not empty* **do**
    $(N, f_k, w) \leftarrow Head(Path)$
    BACKTRACK(N, f_k, w)
    $Path \leftarrow Tail(Path)$
return $UB(Root)$

4.2. Search strategy

The search algorithm we propose above is search strategy agnostic. This comes at the cost of higher complexity and prevents some low-level optimisations, but allows us to compare different search strategies without changing the core algorithm. These comparisons are lacking in the current literature. Previous methods use various search strategies, as discussed in Section 3.2, but only compare entire methods and do not isolate the impact of the search strategy.

4.2.1. Selection

Before we discuss our individual search strategies, we elaborate on the selection procedure in our algorithm and define exactly what we consider a search strategy.

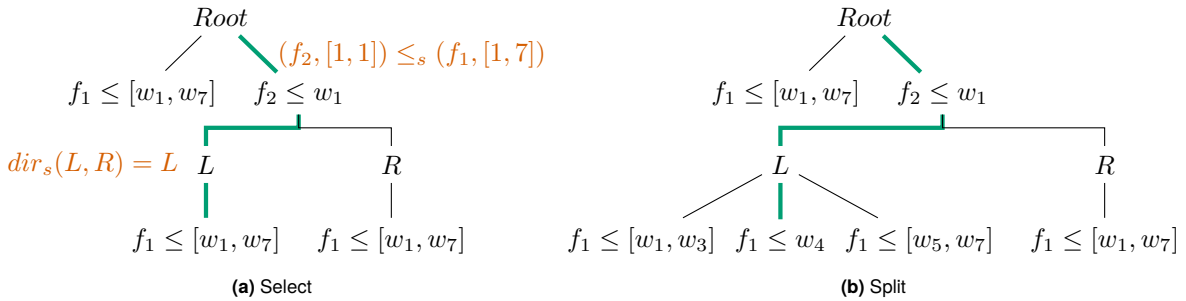
The selection procedure selects a path to a leaf in the partial search graph based on the search strategy. Algorithm 2 uses a priority queue of feature tests for each node $Q(N)$. Starting at the root, it selects a feature test that is not complete from the queue. The algorithm loops until it arrives at a feature test that has not been expanded.

A cornerstone of our algorithm is that our queue stores *intervals* of feature tests. An interval of feature tests $(f_k, [i, j])$ represents all feature tests with feature f_k and any threshold $\{w_k \mid k \in [i, j]\}$. When an interval is selected, we choose the midpoint of the interval for expansion, and the left and right halves are returned to the queue.

We now say that a *search strategy* is a combination of a total order that determines the priority of the feature tests for a selected node, starting from the root, and a function that determines the priority of the

Algorithm 2 SELECT(N) returns a $Path$ to an unexplored split in the search tree.

Initialize empty $Path$
$(f_k, [i, j]) \leftarrow Q(N).pop()$
while $(f_k, [i, j])$ is expanded **do** ▷ When a feature test is expanded, $i = j$
  $Tail(Path) \leftarrow Path$
  $Head(Path) \leftarrow (N, f_k, i)$
  $N_L, N_R \leftarrow Children(N, f_k, i)$
  $N \leftarrow dir_s(N_L, N_R)$
  $(f_k, [i, j]) \leftarrow Q(N).pop()$
$m \leftarrow \lfloor \frac{i+j}{2} \rfloor$
$I(N, f_k, w_m) \leftarrow [i, j]$ ▷ Save the original interval (see Algorithm 5)
if $w - i > 1$ **then** $Q(N).insert((f_k, [i, m-1]))$
if $j - w > 1$ **then** $Q(N).insert((f_k, [m+1, j]))$
$Tail(Path) \leftarrow Path$
$Head(Path) \leftarrow (N, f_k, w_m)$
return $Path$



(a) Select **(b)** Split

Figure 4.2: Single selection procedure in detail. $f_1 \leq [w_1, w_7]$ represents all threshold tests between w_1 and w_7. **(a)** selects the feature test interval based on the search strategy $(\leq_s, dir_s)$, **(b)** selects the middle to expand and splits off the remaining interval.

left and right nodes for a selected feature test. We denote this combination of an ordering and direction selection as a tuple $(\leq_s, dir_s)$.

The total order $\leq_s$ is used for the priority queue in a node $Q(N)$ and determines which feature test should be selected next. dir_s determines if the next node is in the left or right direction. If the left or right node is already complete, dir_s returns the incomplete side. By changing the order $\leq_s$ for the (partially) unexplored candidate splits and the direction dir_s, the algorithm can execute different search strategies.

The selection procedure with the intervals and the search strategy is illustrated in Figure 4.2. The path selected from leaf to root is $(L, f_1, w_4), (Root, f_2, w_1)$.

4.2.2. Goals

First, we consider the goals that a search strategy should achieve. An ideal search strategy would choose the next search node such that:

- **Future information** It is likely to inform maximally on future search decisions. For instance, when the next node has a very high lower bound, we can remove many similar nodes from future search iterations. Similarly, when the next node has a low upper bound, future nodes can be pruned more easily.
- **Anytime** It is likely to lead to a good intermediate solution. In part, this echoes the previous point, as a good intermediate solution informs the search with a low upper bound. However, a good intermediate solution is additionally useful for anytime performance when a proof of optimality is too computationally expensive.

- **Low compute** It does not need much (additional) information to make a decision. The strategy exists to speed up the search, so any additional computation or lookups of information needs to take fewer resources than the benefit of the additional information.
- **Reuse** It is likely to be a repeated subproblem. Because continuous features have a large number of feature tests, it is unlikely that the same sequence repeats unless special care is taken when choosing the thresholds. We disregard this point because we do not cache subproblems (see Section 3.6).

4.2.3. Lower bounds

Having established our goals for the search strategy, we now consider how we can accomplish these. We focus on the future information goal and mention the relevance to other goals in passing.

We postulate that the primary focus for future information should be on informing the lower bounds. To prove optimality, branch-and-bound search needs to traverse the entire search space. Pruning is only achieved when the lower bound exceeds the upper bound. However, finding a reasonable upper bound is fairly easy, as demonstrated by the success of heuristic algorithms such as CART. Rather, it is much harder to obtain a reasonable lower bound, so we focus on this.

There are several methods to obtain lower bounds; we list the ones we consider here.

- **Trivial** First, the trivial lower bound for classification and regression is zero. It is an immediate bound without requiring computation and is a starting point for other bounds. An important property of this bound is that it is an *initial* bound, one that can be established without search.
- **Equivalent points** The equivalent points bound (see Section 3.4) is another initial bound. When multiple instances are indistinguishable by their features, then they all end up with the same prediction. If their label is different, this leads to a minimum error. Since datasets with continuous features are unlikely to have instances with the same feature values, we do not implement this bound.
- **Clustering** We extend the idea of the k-means lower bound for regression (see Section 3.4) to a general *clustering* lower bound. By ignoring possible feature tests and assuming that instances can be partitioned in any way, we relax the problem to a clustering problem. For regression, this is the k-means. For classification, the k majority classes are correctly classified, while all others are misclassified. We elaborate on this bound in Section 4.5.
- **Search exhaustion** When there are no remaining feature tests to consider for a subproblem, the best feature test found so far is the optimal solution. An optimal solution forms a lower and upper bound on the error.
- **Remaining tests** When considering all feature tests for a node that have not been pruned, the lowest lower bound of these remaining tests is a lower bound for the node.
- **Similarity** Similar subproblems have similar solutions, and we can compute a bound based on this similarity (see Section 3.3)
- **Propagation** Since the total error for a subproblem is dependent on its left and right children, any bound for a child can be propagated to its parent and sibling.

The search strategy can influence the latter three bounds. The initial lower bounds and search exhaustion cannot be better informed by previous search decisions. That is, they do not provide a different bound based on the search strategy followed.

The remaining tests bound is influenced by the search strategy in two ways. First, because this bound only applies if a feature test that previously had the lowest lower bound becomes greater. If a different feature test improves its lower bound, the overall lower bound of the node remains the same. Second, we lazily update the bounds in the queue $Q(N)$ so we only know the bounds of the interval at the front of the queue (see Section 4.3). For this reason, we only use the remaining tests bound when we know, due to the search strategy, that the front of the queue has the lowest lower bound.

The propagation bound is also influenced by the search. When searching for the solution of a subproblem, if previous iterations of the search had explored its sibling, then less search is required for this problem.

Finally, we especially highlight the similarity bound. This bound is the main reason why finding an optimal decision tree with continuous features is feasible, as it allows us to obtain a lower bound for entire intervals of feature tests. At the same time, it is highly influenced by the search. Because other lower bounds only affect a single test, we normally do not care about the magnitude of the bound, as long as it is high enough to prune. For the similarity bound, we care about the magnitude of the lower bound because it affects feature tests other than itself. The higher the bound, the more feature tests can be pruned.

Since we care about the height of the lower bound, the search algorithm has to be adapted. Regular branch-and-bound search immediately prunes solutions that are proven to be suboptimal. However, continuing the search might raise the lower bound. Fully exploring the closest unpruned feature test would give at least as much information, but doing so from scratch often requires more effort than improving the lower bound of a test that has already been partially explored. Because of this, we prune a feature test only if it cannot further improve the bound of any other feature test. This is the case when the entire remaining interval can be pruned or when the optimal solution has been found.

4.2.4. Definition of individual search strategies

We have discussed the selection procedure and how it uses a search strategy $(\leq_s, dir_s)$, as well as the goals of the search strategy and how we aim to achieve them. Now, we discuss our choice of search strategies.

We define the order of each search strategy $\leq_s$ by deconstructing it into components. When comparing two intervals of feature tests, the components are each compared in turn. The first component that is not equal is used to order them. Table 4.1 summarises the components used in each search strategy.

The components of each interval of feature tests $(f_k, [i, j])$ are as follows.

Is expanded A feature test *is expanded* when $i = j$ and its left and right nodes are already part of the search graph because it was previously expanded.

Random value Each interval of feature tests is assigned a *random value* when it is first inserted into the queue $Q(N)$ to arbitrarily order them.

Lower bound The *lower bound* for an interval of feature tests is the lowest of all thresholds in the interval and is determined, for example, by the similarity lower bound (see Section 3.3).

Interval size The *size of an interval* is the number of thresholds in $[i, j]$, which is $1 + j - i$.

Feature rank We determine the *feature rank* heuristically for each feature f_k. For classification, the heuristic is the lowest Gini impurity of any threshold. For regression, it is the lowest SSE of any threshold.

Interval start The *start of the interval* i is used for tie-breaks.

Lowest $\mathcal{H}$ Each feature test tracks the lowest heuristic value h of any of its descendants on the search graph, which we refer to as $\mathcal{H}$; we discuss the best-first search heuristics later in this section.

After selecting the feature test, dir_s determines if the next node is in the left or right direction. Table 4.2 lists the direction that each search strategy takes. We use $\mathcal{D}_N$ to refer to the subset of data that remains in the node N after all feature tests in its ancestors. The remainder of this subsection elaborates on the ordering and direction selection of each search strategy.

Depth-first search We consider three variants of depth-first search. The first variant assigns a random priority to each interval of feature tests when it is first inserted into the queue $Q(N)$ and to each node. Since the priority does not change, it keeps selecting the same nodes and feature tests until it is complete, hence being depth-first. We include this variant as a baseline comparison.

With the second variant, we mimic the selection strategy of ConTree (Brița et al. 2025). They exhaustively search each feature in turn, ordered by feature rank. They use a queue for processing the intervals, solve the middle threshold of the interval at the front of the queue, and add the left and right halves back into the queue. This means that the largest intervals are selected first, with the left half before the right half. After selecting the interval, they choose to solve the direction with the largest dataset size first. For classification, this is an indication of the contribution to the total loss, but for regression, that

Table 4.1: Overview of the total order $\leq_s$ defined on an interval of feature tests for each search strategy. Each strategy uses a series of components, the first component is compared first, if the component is equal it continues to the next component.

Order component	DFS-Random	DFS-ConTree	DFS	BFS	AND/OR
Is expanded	1st	1st	1st	2nd	2nd
Random value	2nd				
Lowest lower bound			2nd		1st
Largest interval		3rd	3rd		
Smallest interval					3rd
Feature rank		2nd	4th	3rd	4th
Interval start tie-break		4th	5th	4th	5th
Lowest $\mathcal{H}$				1st	

Table 4.2: Overview of the direction selection dir_s that determines the node to select among N_L and N_R for each search strategy

Search strategy	Direction	Description
DFS-Random	Random value	For comparison
DFS-ConTree	Highest $l(\mathcal{D}_N)$	ConTree uses $\lvert\mathcal{D}_N\rvert$, but leaf error applies to regression
DFS	Highest upper bound	Most room for improvement, or fast pruning
BFS	Lowest $\mathcal{H}$	Descend to globally lowest h
AND/OR	Highest upper bound	Most room for improvement, or fast pruning

is not necessarily the case. This is why instead we use the highest leaf error for the node, $l(\mathcal{D}_N)$, as the direction heuristic.

For the final variant, which we refer to simply as *DFS*, we make several changes that may give the lower bounds more information. First, we use the lower bound to guide the selection of the intervals. In this way, we deprioritise intervals with known high error and, when none of the current intervals are expanded, the next feature test will have the lowest lower bound of the remaining feature tests and thus be a lower bound for the node by the *remaining tests* lower bound.

The second change for DFS is that we interleave feature tests with different features, instead of committing to a single feature first. This reduces the impact of an incorrect initial feature choice when the feature ranking fails to provide the best feature.

Finally, we change the chosen direction to the node with the highest upper bound. When the node has a higher upper bound, finding a better solution for this node will likely be a bigger improvement, and not finding a better solution results in a high lower bound. Note that the upper bound might change across search iterations, so the direction to explore might change before the initial choice is fully solved. However, since the choice of the feature test does not change until it is solved, at most 2^d nodes are active at once. This is more than the d nodes during regular depth-first search, but not significantly so for our assumption of a small d.

Global best-first search For global best-first search, our algorithm can emulate a global queue. Since we want to select the feature test with the globally lowest heuristic h, we maintain the lowest heuristic score of any of the descendants of a node and denote this value $\mathcal{H}$. We update $\mathcal{H}$ during backtracking. The nodes and feature tests are then ordered by $\mathcal{H}$, so that the feature test at the end of the selected path always has the globally lowest heuristic. We define and provide an overview of all the heuristics used in Table 4.3, and discuss these more in-depth below.

We use the different heuristic functions discussed in Section 3.2. h_{LB} uses the lowest lower bound of the objective directly, $h_{Curiosity}$ uses the number of remaining instances to prioritise more impactful nodes, and finally h_{GOSDT} subtracts instead of divides the number of remaining instances.

The GOSDT heuristic was not discussed in the papers that make use of it, so the original reasoning for it is unclear. We offer our reasoning for this heuristic as follows. We approach the heuristic in a highest

Table 4.3: Best-first search heuristics. The lowest value of h has the highest priority. LB is the current lower bound on the objective for the feature test(s). $|\mathcal{D}_N|$ is the number of remaining instances for the node N. *Regression discussed in text.

BFS Heuristic	Definition	Intuition
h_{LB}	LB	Lowest minimum error first
$h_{Curiosity}$	$\frac{LB}{\lvert\mathcal{D}_N\rvert}$	Lowest minimum error first, weighted by impact on root
h_{GOSDT}	$LB - \lvert\mathcal{D}_N\rvert$	Most instances that may still be classified correctly first*
h_{Small}	$\lvert\mathcal{D}_N\rvert$	Smallest node first for quick bounds
h_{Big}	$-\lvert\mathcal{D}_N\rvert$	Largest node first for high bounds
$h_{SmalltLB}$	$\frac{LB}{100000} + \lvert\mathcal{D}_N\rvert$	Smallest node first for quick bounds, break ties with LB
h_{BigtLB}	$\frac{LB}{100000} - \lvert\mathcal{D}_N\rvert$	Largest node first for high bounds, break ties with LB
$h_{LBtSmall}$	$LB + \frac{\lvert\mathcal{D}_N\rvert}{100000}$	Lowest minimum error first, break ties with smaller nodes
h_{LBtBig}	$LB - \frac{\lvert\mathcal{D}_N\rvert}{100000}$	Lowest minimum error first, break ties with larger nodes
$h_{LB\&Small}$	$LB + \lvert\mathcal{D}_N\rvert$	Low minimum error and small node size in equal measure
h_{Random}	Random priority	For comparison

value first manner. For our context, that means we negate the heuristic $-h_{GOSDT}(N) = |\mathcal{D}_N| - LB$. We see that for classification, this is the number of instances that can be correctly classified. For regression, the intuition is less clear. Either of the two terms could dominate the equation depending on the scale of the dataset, causing the other to act as a tiebreaker. Zhang et al. (2023) use the GOSDT heuristic in the regression setting and, perhaps to reduce the scale dependence of this heuristic, normalise each instance label by the root of the sum of squares (RSS) of the dataset.[1]

We introduce several other heuristics based on the lower bound and the number of remaining instances to compare against. First, to see if the lower bound contributes meaningfully to the heuristic, we include heuristics that leave it out. Second, to test whether it is better to explore nodes with a lesser or greater number of instances remaining, we include heuristics that prioritise smaller, instead of bigger, nodes. Third, we test whether each term contributes equally to the heuristic or if one is more important, and the other acts as a tiebreaker. To test this, we vary the scale of each term. Finally, we also compare with the random heuristic that assigns a random priority to each interval of feature tests.

AND/OR best-first search As discussed in the preliminaries, AND/OR search always selects a feature test that is part of the tree with the lowest lower bound. This allows the *remaining tests* lower bound to be fully exploited, because only improving this lower bound has a chance of increasing the overall lower bound for the node. In addition, the priority queue of feature tests for each node allows efficient lookup of the feature test with the lowest lower bound, which is necessary to use the remaining tests lower bound in practice. We do note that the remaining tests lower bound is not the only lower bound, and the similarity lower bound might increase the lowest lower bound indirectly by searching a nearby threshold that itself is not the lowest.

At some point during the search, the lower bound of multiple feature tests might be equal. For example, at the start of the search, the lower bounds for all feature tests are zero. For this reason, we use several tiebreakers. First, to reduce memory usage, we prefer nodes that have already been expanded.

Second, we prefer shorter threshold intervals. Our reasoning for short intervals is that these typically have a more refined bound than large intervals, and since they also have the lowest lower bound, they may be closer to the true solution.

Finally, we break the remaining ties by the best feature ranked heuristically and then by the start of the threshold interval for a greedy start to the search and a deterministic search order, respectively.

So far, we have covered which feature test AND/OR search selects for each node, but not how the choice is made between the left and right nodes. For this, we select the node with the highest upper

[1]The heuristic and the scaling were not discussed in their paper, but are included in the reference implementation, `https://github.com/ruizhang1996/optimal-sparse-regression-tree-public`, cc7961b44fc8d80b73fcda0348d1f3deac135391

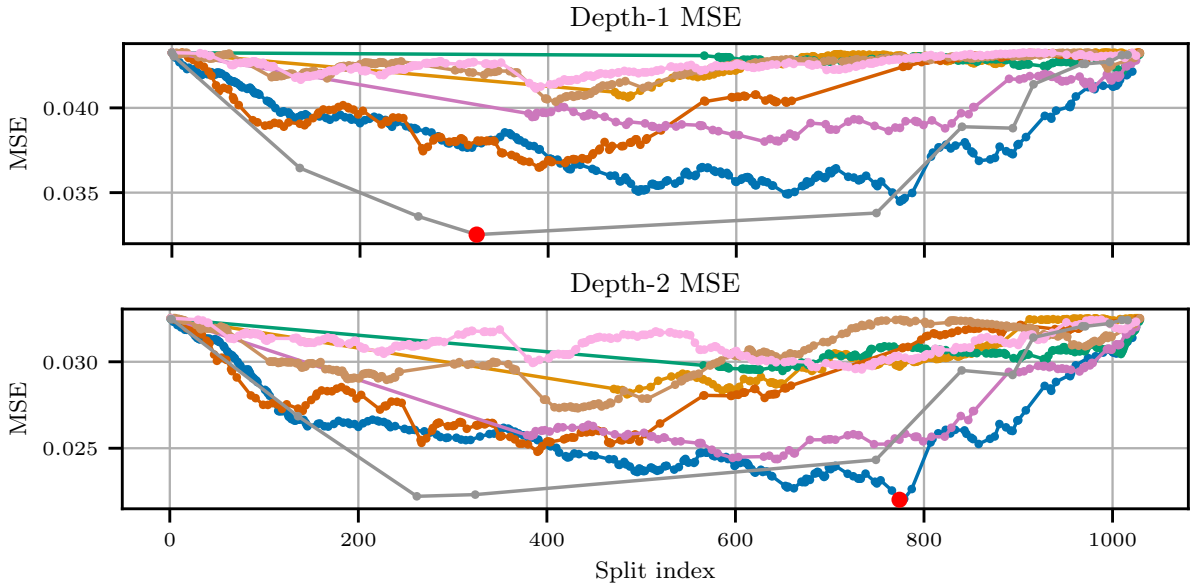


Figure 4.3: The (optimal) MSE for a fixed feature test in the root. Each line is a different feature, each point is a different threshold. It shows that the depth-1 error (used as a heuristic) is indicative of the depth-2 error, but the optimal depth-1 feature test (red dot) is not the optimal depth-2 feature test. *concrete* is used here as an illustrative dataset.

bound for the same reason as we do in DFS, a higher impact on the overall bound of the feature test.

Limited discrepancy search Limited discrepancy search (LDS) explores feature tests closer to the heuristically best solution first. This is applied for a fixed set of binary feature tests in Kiossou et al. (2022). They perform a series of searches, each with an increasing discrepancy budget. They count the discrepancies for a feature test as its rank for a given heuristic. For each search, they use depth-first search until the accumulated discrepancy is greater than the budget.

Since our algorithm does not perform multiple searches iteratively, we implement LDS differently. We use the framework of our global best-first search, and use the number of discrepancies, including the discrepancy accumulated by its ancestors, as the global heuristic for a feature test.

We illustrate the benefit of LDS for anytime performance with Figure 4.3, which shows an illustrative example of the error for all possible root feature tests in the *concrete* dataset. It shows that a low heuristic (depth-1 error) is indicative of a low error at higher depths (depth-2 error). This is why greedy methods such as CART are effective and why we first search for trees near the heuristic solution with LDS.

However, for our case with continuous feature tests, we argue that it may be better to assign the discrepancy value in a different way. Our primary pruning method is to prune intervals of feature tests with the similarity lower bound. As Figure 4.3 shows, there are many feature tests with similar thresholds that also have similar heuristics. If we were to explore these in turn, the similarity lower bound cannot prune many feature tests since it relies on a large difference between the upper and lower bounds.

Therefore, it may be better to explore a few different intervals for different features than many similar thresholds for the same feature. We choose to count the number of discrepancies by the feature rank, which is the lowest Gini impurity (for classification) or SSE (for regression) of any threshold for that feature, plus the number of times the threshold interval has been split. For example, for the fifth best feature f_k with a thousand thresholds with the heuristically best threshold w_{300}, $(f_k, [1, 1000])$ and $(f_k, [300, 300])$ would both have a discrepancy of $4+0=4$, while $(f_k, [1, 299])$ and $(f_k, [301, 1000])$ would have a discrepancy of $4+1=5$.

Because LDS relies on the first tree to be the heuristically best one, we pick the first split of each feature to be the threshold with the lowest Gini impurity or SSE, before resuming with our regular approach of splitting the interval in the middle. Note that if we continued to select the lowest Gini impurity or SSE,

Algorithm 3 EXPAND(N, f_k, w) expands a split. UseLeftRight is enabled when the left-right solver is used (Section 4.4).

if *UseLeftRight* $\wedge\ d_N = 2$ **then**
  Let N_L and N_R be the left and right children of N with feature test $f_k \leq w$
  $LB(N_L), LB(N_R) \leftarrow UB(N_L), UB(N_R) \leftarrow$ SOLVELEFTRIGHT$(\mathcal{D}_N, f_k, w)$
else
  $N_L \leftarrow$ CONSTRUCTNODE$(\mathcal{D}_N(f_k \leq w), d_N - 1)$
  $N_R \leftarrow$ CONSTRUCTNODE$(\mathcal{D}_N(f_k > w), d_N - 1)$
$Children(N, f_k, w) \leftarrow N_L, N_R$

the explored feature tests would be extremely similar.

4.3. Expanding and backtracking

In Section 4.1, we have discussed the main steps of Algorithm 1 and the selection procedure. In this section, we discuss the remaining expansion and backtracking procedures in greater detail. We defer the discussion of specialised solvers for shallow trees to Section 4.4 and suggest the reader to ignore the parts of the algorithm conditioned on $UseD1$, $UseD2$, and $UseLeftRight$ until then.

The expansion procedure (Algorithm 3) constructs the left and right child nodes based on the selected feature test. It uses Algorithm 4 to construct each node. When the node is constructed, we set the initial bounds. The upper bound is set to the leaf error $l(\mathcal{D})$, and we use the clustering lower bound (see Section 4.5) to set the initial lower bound. If the remaining depth is shallow enough, we use the exhaustive solvers for shallow trees to find the exact solution immediately. Otherwise, we add all useful feature tests to the priority queue $Q(N)$ for further search.

Not all feature tests are useful. For example, if an instance satisfies the feature test $f_1 \leq w_2$, then it trivially also satisfies the feature test $f_1 \leq w_1$, so the second feature test is not useful in any node part of the left subtree of the first feature test. Therefore, we only consider feature tests that partition the remaining instances such that the left and right sides are not empty.

Another situation in which a feature test is not useful is when there are multiple feature tests for the same feature that have zero error on one side. In that case, the feature tests that achieve zero error while covering fewer instances are not useful. For example, if the errors on the left of $f_1 \leq w_1$ and $f_1 \leq w_2$ are both zero, then $f_1 \leq w_1$ is not useful. We remove these feature tests when constructing the node.

After expanding the feature test, we backtrack the refined bound to its ancestors with Algorithm 5. For the upper bound, this is straightforward; if the upper bound of this feature test is the best so far, then we update the upper bound of this node. The lower bound is more complicated, as the new lower bound for the feature test may impact several other (intervals of) feature tests in the queue by the similarity lower bound.

The lower bound for a feature test is the sum of a lower bound for the left subtree and a lower bound for the right subtree. For each, we can use either the subset or the superset bound (see Section 3.3). For example, consider the lower bound for the left subtree of the feature test $f_1 \leq w_2$. With the subset bound, we can directly use the lower bound of a left subtree with a lower threshold, such as $f_1 \leq w_1$. With the superset bound, we can use the lower bound of a left subtree with a higher threshold, such as $f_1 \leq w_3$, but then we need to subtract the maximum error for the instances in between the thresholds $e^{max}(\mathcal{D}, \mathcal{D}(w_2 < f_1 \leq w_3))$.

Previous approaches that use the similarity bound to prune entire intervals of thresholds can use the closest known optimal solution that is suitable, since they compute the optimal solution immediately. However, because we do not commit entirely to a selected feature test, we only know lower bounds at varying levels of refinement. This means that, for example, we might have a better lower bound for the left subtree of $f_1 \leq w_1$ than for the left subtree of $f_1 \leq w_2$, even if we know that when the search is finished, the second must be higher.

Algorithm 4 ConstructNode$(\mathcal{D}, d)$ creates a new node for the given dataset and depth. *UseD1* and *UseD2* are enabled when the depth-one, respectively depth-two solver are used (see Section 4.4).

Let N be the new node
$\mathcal{D}_N \leftarrow \mathcal{D}$
if *UseD2* $\wedge\, d = 2$ **then**
  for $f \in \mathcal{F}$ **do**
    $\theta_L, \theta_R \leftarrow$ D2Solve$(\mathcal{D}, f)$
    $LB(N) \leftarrow UB(N) \leftarrow \min(l(\mathcal{D}), \theta_L + \theta_R + \lambda)$ ▷ $l(\mathcal{D})$ may be lower if $\lambda > 0$.
  return N
if *UseD1* $\wedge\, d = 1$ **then**
  $UB(N) \leftarrow \min_{k,i}(l(\mathcal{D}(f_k \leq w_i)) + l(\mathcal{D}(f_k > w_i)) + \lambda)$ ▷ In $\mathcal{O}(|\mathcal{D}|)$, similar to Algorithm 6
  $LB(N) \leftarrow UB(N) \leftarrow \min(UB(N), l(\mathcal{D}))$
  return N
$UB(N) \leftarrow l(\mathcal{D})$
$Q(N) \leftarrow \emptyset$ ▷ Min-first priority queue using $\leq_s$
if $UB(N) > \lambda \ \wedge d > 0$ **then**
  for $f \in \mathcal{F}$ **do**
    $U \leftarrow \{u_1 < u_2 < ... < u_k \mid u_i = x^f, (x, y) \in \mathcal{D}\}$
    $W_N^f \leftarrow \{w_i = \frac{u_i + u_{i+1}}{2} \mid i \in [1, k) \wedge |\mathcal{D}(f \leq w_i)| > 0 \wedge |\mathcal{D}(f > w_i)| > 0\}$
    $w_{min} \leftarrow \max\{w \mid l(\mathcal{D}(f \leq w)) = 0\}$
    $w_{max} \leftarrow \min\{w \mid l(\mathcal{D}(f > w)) = 0\}$
    $W_N^f \leftarrow W_N^f \cap [w_{min}, w_{max}]$
    $Q(N).insert(f, [1, |W_N^f|])$
  $LB(N) \leftarrow \min_{1 \leq k \leq \min(2^d, 1 + \lfloor \frac{UB(N)}{\lambda} \rfloor)}$(Clustering$(\mathcal{D}, k) + k\lambda$)
else
  $LB(N) \leftarrow UB(N)$
return N

For this reason, we maintain four structures to calculate the similarity lower bound: the best lower bound for a left-subtree left of each threshold LB_{BL}, the closest lower bound for a left-subtree right of each threshold LB_{CL}, and vice versa with LB_{BR} and LB_{CR} for right-subtrees. We use a B-tree for each of these to update and query them efficiently.

After updating the structures, we only use them to update the bounds for the feature test at the front of the queue, repeating if the update causes a change in order. In this way, we can lazily update the lower bound of the feature tests and avoid updating the entire queue for every iteration of the search.

When updating the upper bound for the feature test, we impose a limit on the tightness of the bound. We do this because, similar to Brița et al. (2025), we observe that setting the upper bound tightly severely reduces the effectiveness of the similarity bound. When the upper bound is too tight, pruning occurs before the lower bound is good enough to also prune neighbouring feature tests. Unlike Brița et al. (2025), who use a heuristic that works well in practice, we set the upper bound so that it exactly prunes when, by similarity, the entire original interval $I(N, f_k, i)$ can be pruned. Since the margin m for this may be unreasonably large, we also use the upper bound of the node as an upper bound for the feature test.

For computing the margins, we often query e^{max}. For classification, this is simply the size of the interval and therefore runs in constant time. However, for regression, it is a sum over the instances in the interval. To compute this quickly, we construct a segment tree for this sum in each node. That is, for each node and feature f_k, we can compute $e_r^{max}(\mathcal{D}_N, \mathcal{D}_N(f_k \geq i \ \wedge \ f_k \leq j))$ in $\mathcal{O}(\log |\mathcal{D}_N|)$ time.

Finally, when the bounds at the front of the queue have been updated and it has the lowest remaining lower bound, then it is a lower bound for the node. We do not iterate over the entire queue to check if it is minimal, but we know this is the case for some search strategies. In particular, for AND/OR best-first search, this is always the case. For our depth-first search, this is the case when the next feature test is not expanded.

Algorithm 5 BACKTRACK(N, f_k, w) updates the node with the refined bounds of feature test $f_k \leq w$.

$N_L, N_R \leftarrow Children(N, f_k, w)$
$UB(N) \leftarrow \min(UB(N), UB(N_L) + UB(N_R) + \lambda)$
▷ *Update structures with new lower bound information*
$LB_{BL}(N, f_k, x) \leftarrow \max(LB_{BL}(N, f_k, x), LB(N_L)) \quad \forall x \geq w$ ▷ Average $O(\log n)$ using B-tree
$LB_{BR}(N, f_k, x) \leftarrow \max(LB_{BR}(N, f_k, x), LB(N_R)) \quad \forall x \leq w$ ▷ Average $O(\log n)$ using B-tree
$LB_{CL}(N, f_k).insert(w, LB(N_L))$
$LB_{CR}(N, f_k).insert(w, LB(N_R))$
$i \leftarrow j \leftarrow w$
loop
  ▷ *Determine lower bound for the left subtree with LB_{BL} or LB_{CL}, vice versa for the right subtree*
  $a, lb_{cl} \leftarrow LB_{CL}(N, f_k).closestRightOf(i)$
  $b, lb_{cr} \leftarrow LB_{CR}(N, f_k).closestLeftOf(j)$
  $l \leftarrow \max(LB_{BL}(N, f_k, i), lb_{cl} - e^{max}(\mathcal{D}_N, \mathcal{D}_N(f_k > i \wedge f_k \leq a)))$
  $r \leftarrow \max(LB_{BR}(N, f_k, j), lb_{cr} - e^{max}(\mathcal{D}_N, \mathcal{D}_N(f_k > b \wedge f_k \leq j)))$
  $LB(N, f_k, [i, j]) \leftarrow l + r + \lambda$
  if $(f_k, [i, j])$ *is expanded* **then**
    ▷ *Update $UB(N_L)$ using $UB(N)$ and a lower bound for N_R. Add a margin m to prevent pruning until neighbouring feature tests can be pruned by the similarity lower bound, vice versa for $UB(N_R)$.*
    $N_L, N_R \leftarrow Children(N, f_k, i)$
    $i_o, j_o \leftarrow I(N, f_k, i)$ ▷ The original interval, see Algorithm 2
    $m_l \leftarrow e^{max}(\mathcal{D}_N, \mathcal{D}_N(f_k \geq i_o \wedge f_k < i))$
    $m_r \leftarrow e^{max}(\mathcal{D}_N, \mathcal{D}_N(f_k > i \wedge f_k \leq j_o))$
    $m \leftarrow \max(m_l, m_r)$ ▷ Limit tightness of UB so $I(N, f_k, i)$ can be pruned
    $UB(N_L) \leftarrow \min\{UB(N_L), UB(N), UB(N) - r + m\}$
    $UB(N_R) \leftarrow \min\{UB(N_R), UB(N), UB(N) - l + m\}$
  if $l + r + \lambda < UB(N)$ **then**
    $Q(N).insert(f_k, [i, j])$
    if $Q(N).peek() = (f_k, [i, j])$ **then break**
  if $Q(N)$ *is empty* **then**
    $LB(N) \leftarrow \max(LB(N), UB(N))$
    break
  $(f_k, [i, j]) \leftarrow Q(N).pop()$
if $LB(N, Q(N).peek())$ *is minimal in* $Q(N)$ **then** ▷ Only if this check is $\mathcal{O}(1)$
  $LB(N) \leftarrow \max(LB(N), LB(N, f_k, [i, j]))$

For clarity, from Algorithm 5 we exclude that we shrink the intervals $I(N, f_k, i)$ and $[i, j]$ when a range at the beginning or end can be pruned by similarity or because a better zero-cost solution has been found, as described in detail by Brița et al. (2025). However, we note that this is only useful for selecting the midpoint in Algorithm 2 and reducing the tightness limit of the upper bound.

4.4. Specialised solver for shallow trees

The majority of the computation is done at the bottom of the search tree. As covered in Section 3.5, a specialised solver for trees of depth two provides a large speedup for trees with binary features. For continuous features, solving the left and right parts of a depth-two tree for a fixed feature test at the root node is effective. We provide two implementations of solvers for shallow trees and one additional concept of a solver in pseudocode.

Left-right solver We generalise the $\mathcal{O}(|\mathcal{D}||\mathcal{F}|)$ algorithm from Brița et al. (2025) to simultaneously solve the left and right depth-one subtrees of a depth-two tree to regression in Algorithm 6. The difference from the original algorithm for classification is what we keep track of to calculate the leaf error for each split. For classification, this is the number of instances of each class, and we calculate the leaf error by subtracting the (correctly classified) majority class from the total number of instances. Our

adaptation to regression uses three running sums of $\sum y$, $\sum y^2$, and $|\mathcal{D}|$. We then use the function $SSE(\sum y, \sum y^2, |\mathcal{D}|) = \sum y^2 - \frac{(\sum y)^2}{|\mathcal{D}|}$ to compute the SSE using three running sums; see Appendix A for a complete derivation.

Algorithm 6 SolveLeftRight($\mathcal{D}, f_1, w$) solves the left and right side of the tree for a given feature f_1 and its threshold w, for regression. Adapted from (Brița et al. 2025).

$T_L, T_R \leftarrow (0, 0, 0)$ ▷ Totals for the left and right subtree
for $(x, y) \in \mathcal{D}$ **do**
  if $x_{f_1} \leq w$ **then**
    $T_L \leftarrow T_L + (y, y^2, 1)$
  else
    $T_R \leftarrow T_R + (y, y^2, 1)$
$\theta_L \leftarrow SSE(T_L), \theta_R \leftarrow SSE(T_R)$ ▷ Current best solutions for left and right subtree
for $f_2 \in \mathcal{F}$ **do**
  $T_{LL}, T_{RL} \leftarrow (0, 0, 0)$ ▷ Totals of the left-left and the right-left leaves
  for $(x, y) \in \mathcal{D}$ *sorted by* f_2 **do**
    if $x_{f_1} \leq w$ **then**
      $T_{LL} \leftarrow T_{LL} + (y, y^2, 1)$
      $\theta_L \leftarrow \min(\theta_L, SSE(T_{LL}) + SSE(T_L - T_{LL}) + \lambda)$
    else
      $T_{RL} \leftarrow T_{RL} + (y, y^2, 1)$
      $\theta_R \leftarrow \min(\theta_R, SSE(T_{RL}) + SSE(T_R - T_{RL}) + \lambda)$
  if $\theta_L \leq \lambda \wedge \theta_R \leq \lambda$ **then break**
return θ_L, θ_R

Depth-one solver The performance advantage of solving the left and right depth-one trees simultaneously has not been experimentally verified, so we also implement a similar algorithm that directly solves depth-one subtrees as a baseline. We omit the pseudocode because it is similar to Algorithm 6.

Depth-two solver We also propose a concept for a full depth-two solver that does not fix the threshold at the root node. We show the pseudocode in Algorithm 7 for regression, but note that it can easily be adapted to classification by counting the number of instances in each class and computing the leaf error accordingly.

The main idea of the algorithm is to go through the depth-one tree as in Algorithm 6, but to keep track of the left and right errors for each threshold w_i, instead of one fixed threshold test.

We make two improvements that may allow the full depth-two solver to perform better than calling Algorithm 6 for each threshold. First, we store the totals for the left-left and right-left leaves in a Fenwick tree (Fenwick 1994). This allows us to update the totals for all thresholds in logarithmic time.

Second, we update the costs sparsely using similarity. Updating the lowest costs requires linear time over the interval $[i, j]$ if done naively. This would result in an average and worst-case complexity of $\mathcal{O}(|\mathcal{F}||\mathcal{D}||W^{f_1}|)$ for Algorithm 7. However, for adjacent thresholds w_i and w_{i+1}, we know the maximum number of instances that could have changed by similarity. We use this in Algorithm 8 to update the lowest costs sparsely. This has the same worst-case complexity, but may be superior in the average case.

The depth-two solver is a concept and has not been implemented, which leaves the performance impact unclear. The full depth-two solver uses similarity to sparsely update the current best costs, but normal search can also take advantage of similarity to find the test in the root node.

4.5. Clustering lower bound

We can relax the problem of finding a decision tree by ignoring the features, reducing the problem to an optimal clustering of instances in the leaves. The solution to this relaxed problem is a lower bound, as reintroducing the restriction of the feature values cannot improve the loss. For regression, this is

Algorithm 7 D2SOLVE$(\mathcal{D}, f_1)$ finds the optimal threshold for f_1, and solves the left and right side of the tree.

$T \leftarrow (0,0,0)$ ▷ Totals for whole tree
$T_i^L, T_i^R \leftarrow (0,0,0) \quad \forall i \leq |W^{f_1}|$ ▷ Totals for left and right subtree for split point i
$i \leftarrow 1$
for $(x,y) \in \mathcal{D}$ *sorted by* f_1 **do**
  $i_{new} \leftarrow \text{argmin}_i\{w_i \in W^{f_1} \mid x_{f_1} \leq w_i\}$
  if $i \neq i_{new}$ **then**
    $T_i^L \leftarrow T$
    $i \leftarrow i_{new}$ ▷ This is equal to $i \leftarrow i+1$
  $T \leftarrow T + (y, y^2, 1)$
$T_i^R \leftarrow T - T_i^L \quad \forall i \leq |W^{f_1}|$
$\theta_i^L \leftarrow SSE(T_i^L), \theta_i^R \leftarrow SSE(T_i^R) \quad \forall i \leq |W^{f_1}|$ ▷ Current best solution for each f_1 split
for $f_2 \in \mathcal{F}$ **do**
  $T_i^{LL}, T_i^{RL} \leftarrow (0,0,0) \quad \forall i \leq |W^{f_1}|$ ▷ Totals of the left-left and the right-left leaves
  for $(x,y) \in \mathcal{D}$ *sorted by* f_2 **do**
    $s \leftarrow \text{argmin}_i\{w_i \in W^{f_1} \mid x_{f_1} \leq w_i\}$
    $T_i^{LL} \leftarrow T_i^{LL} + (y, y^2, 1) \quad \forall i \geq s$ ▷ $O(\log |W^{f_1}|)$ with Fenwick tree
    $T_i^{RL} \leftarrow T_i^{RL} + (y, y^2, 1) \quad \forall i < s$ ▷ $O(\log |W^{f_1}|)$ with Fenwick tree
    $\text{UpdateLowestCosts}(\theta^{\text{L}}, [\text{s}, |\text{W}^{\text{f}_1}|], \text{T}^{\text{L}}, \text{T}^{\text{LL}})$
    $\text{UpdateLowestCosts}(\theta^{\text{R}}, [0, \text{s}-1], \text{T}^{\text{R}}, \text{T}^{\text{RL}})$
$i_{best} \leftarrow \text{argmin}_i\, \theta_i^L + \theta_i^R$
return $\theta_{i_{best}}^L, \theta_{i_{best}}^R$

the k-means, as discussed in the preliminaries Section 3.4. However, to the best of our knowledge, this has never been applied to classification. We generalise the idea of the k-means lower bound to classification and call this the *clustering lower bound*.

For classification, an optimal clustering of the instances, over k leaves, classifies the k majority classes correctly. This is a useful lower bound for multi-class classification when $k < |\mathcal{Y}|$. In particular, it is not useful for binary classification.

We can bound the maximum number of leaves k by the remaining maximum depth and the regularisation parameter: $k \leq \min(2^d, 1 + \lfloor \frac{UB}{\lambda} \rfloor)$. However, even if we may use k leaves, the $(k-1)\lambda$ penalty for using all of these may be large, so we incrementally check all clusterings from one to k.

In the following subsections, we introduce an exactness test and a depth-one solver that expand on the idea of the clustering lower bound. However, we have not implemented these ideas.

Algorithm 8 $UpdateLowestCosts(\theta, [i,j], T, T^L)$ using a similarity bound.

$Q = \{[i,j]\}$
$worst \leftarrow \max_{(x,y)\in\mathcal{D}} e^{max}(\mathcal{D}, \{(x,y)\})$
while $|Q| > 0$ **do**
  $[i,j] \leftarrow Q$.pop()
  $w \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  $\Theta \leftarrow SSE(T_w^L) + SSE(T_w - T_w^L) + \lambda$
  $\Delta \leftarrow \Theta - \theta_w$
  if $\Delta < 0$ **then**
    $\theta_w \leftarrow \Theta$
    Q.push$(\{[i, w-1], [w+1, j]\})$
  else
    $skip \leftarrow 1 + \lfloor \frac{\Delta}{worst} \rfloor$ ▷ For simplicity, assume each point has unique feature values
    Q.push$(\{[i, w-skip], [w+skip, j]\})$

4.5.1. Exactness test

In addition to using the clustering as a lower bound on the error, we can find an upper bound by searching for a decision tree with leaves that most closely reproduce the reference cluster. If feature tests can be found such that the tree matches the clustering exactly, we have found the optimal solution.

In general, it is NP-hard to find the feature tests that produce the closest matching clustering, as this is equivalent to finding an optimal classification tree where the label of each instance is its cluster. This is why we provide the following sufficient (but not necessary) heuristic procedure to find a decision tree that perfectly matches the clustering. It runs in linear time under the normal assumption that $k \ll |\mathcal{D}|$.

Theorem 4.1. Let the k-clustering C, as found by the clustering lower bound, be a partition of $\mathcal{D}$. Take the minimum and maximum value for a feature $f \in \mathcal{F}$ in a cluster $c \in C$ as $m_c^f = \min_{(x,y)\in c} x^f$ and $M_c^f = \max_{(x,y)\in c} x^f$. If there is a single feature that isolates all clusters, then the clustering is an exact solution. In other words, if $\exists f \in \mathcal{F}, \forall c, k \in C, (k = c \vee M_c^f < m_k^f \vee m_c^f > M_k^f)$, then the clustering lower bound is also an upper bound for a decision tree of k leaves.

Proof. We show that this condition is sufficient by construction. If $k = 1$, then a decision tree without any branches suffices, and we are done. Otherwise, take $f \in \mathcal{F}$ to be the feature for which the proposition holds. Let $c_i \in C$ be the cluster with the ith lowest maximum feature value $M_{c_i}^f$. Let w_{c_i} be the minimum threshold that includes all instances of c_i, and let $w_{c_0} = -\infty$. Since the proposition holds, $\forall c_i \in C, (c_i = \mathcal{D}(w_{c_{i-1}} < f \leq w_{c_i}))$. We construct the decision tree using a divide-and-conquer approach to ensure that the resulting tree has a maximum depth of $\lceil \log_2(k) \rceil$. Let $j = \lfloor \frac{k}{2} \rfloor$. We choose the feature test in the root as $f \leq w_{c_j}$. If $k = 2$, then we are done. Otherwise, recurse to construct the left subtree for the j-clustering $\{c_i \mid i \leq j\}$, and the right subtree for the $\lceil \frac{k}{2} \rceil$-clustering $\{c_i \mid i > j\}$. □

For two reasons, the proposition in Theorem 4.1 is not necessary for a perfect tree to exist. First, it does not check for the existence of a tree that uses feature tests of multiple features. Second, it only tests against a single minimal reference clustering, while multiple minimal clusterings may exist.

As a curiosity, we mention that this might seem to imply that finding a perfect regression tree for a single continuous feature is not NP-hard. Using the DP approach described in Section 3.4, it may seem that we can iterate over all minimal k-clusterings. However, we cannot, as the DP approach fixes the ordering based on the label value. If each label is unique, then we can check if a perfect regression tree exists in polynomial time, but in that case it is trivial, since a perfect tree exists if and only if $|\tau| = |\mathcal{D}| - 1$.

4.5.2. Depth-one solver

The clustering lower bound can be used to obtain an initial bound on the problem without search. However, in many cases, this lower bound is lower than the actual solution because the feature tests cannot be used to obtain this clustering in the leaves.

To refine this bound and make it more closely reflect the actual solution after branching, we can check the clustering lower bound for the left and right side of all possible feature tests, before committing to select any of the feature tests for the search. The lowest among these is now a lower bound for the node, and the search is better informed about which feature tests it should select. This depth-one clustering lower bound is guaranteed to give a lower bound for the node that is at least as good as the clustering lower bound without branching, but has the potential to be much higher. The question now is whether we can do this check faster than simply expanding all of the feature tests.

For classification, since the k-clustering only requires the frequency count of each class, this can be done in linear time by looping through the instances sorted by each feature value, similar to the depth-one solver for the loss.

For regression, achieving this efficiently remains an open question. The existing DP approach to k-means clustering has optimal solutions to subproblems, but for the instances sorted by the label value (see Section 3.4), which does not help us since we would like subproblems ordered by each feature value. If the depth-one clustering lower bound can be found for regression with a runtime that is less than quadratic in the size of the dataset, we expect this to be a substantial improvement.

5

Experiments

We evaluate our method by experimentally comparing it to greedy heuristics for generalisation and to the state-of-the-art methods in optimal decision trees with continuous features for runtime. Our experiments answer the following research questions:

1. **Does our algorithm, *CODTree*, find optimal classification and regression trees faster than the state-of-the-art?**
   For regression, our method is significantly faster to prove optimality than Quant-BnB at depths two and three, and, to the best of our knowledge, the only method that proves optimality for trees of depth four. For classification, our method is significantly faster than Quant-BnB and comparable in performance to ConTree. However, our search has up to four orders of magnitude less graph expansions than ConTree with diminishing returns at higher depths, this suggests that their code is more optimised while our search strategy is better.
2. **What search strategy is best for finding optimal classification and regression trees in terms of time to optimal solution and anytime performance?**
   Global best-first search with a heuristic that selects based on smaller subproblems first and better lower bounds second is the best in terms of both the time to optimality and the anytime performance. This search strategy is significantly better than many others, often finding a tree in a few seconds when many other strategies quickly run out of memory. In particular, search strategies that do not prioritise smaller subproblems invariably run out of memory due to the large branching factor in the search graph for continuous features, for $d > 2$.
3. **What is the impact of the specialised solver for shallow trees on the time to prove optimality?**
   For the datasets that can be solved without a specialised solver, the left-right solver and the depth-one solver have a geometric mean speedup of 77.4x and 36.6x, respectively. Therefore, we conclude that the left-right solver is superior.
4. **What is the impact of the clustering lower bound on the time to prove optimality?**
   We do not see a significant difference by including this lower bound, with a geometric mean reduction of two percent in the graph expansions, and a three percent increase in the time to optimality. A depth-one solver of the clustering lower bound can improve this bound in the root.
5. **Do optimal regression trees generalise better to new data than greedy regression trees with models of equivalent complexity (maximum depth)?**
   At all depths in our experiment (two to four), CODTree generalises better than CART on more datasets. At depth three, CODTree is never worse than CART. However, we have insufficient data to reject the null hypothesis.

Table 5.1: Datasets used in experiments with the number of instances $|\mathcal{D}|$, features $|\mathcal{F}|$, feature tests $\sum_f |W^f|$, and unique labels $|\mathcal{Y}|$.

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\sum_f \lvert W^f\rvert$	$\lvert\mathcal{Y}\rvert$	Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\sum_f \lvert W^f\rvert$	$\lvert\mathcal{Y}\rvert$
Classification					*Regression*				
avila	20867	10	41110	12	casp	45730	9	298346	15903
bank	1372	4	5016	2	concrete	1030	8	1517	845
bean	13611	16	211966	7	energy	19735	28	107798	92
bidding	6321	9	12528	2	fish	908	6	1810	827
eeg	14980	14	5404	2	gas	36733	10	142100	4447
fault	1941	27	19286	7	grid	10000	12	119988	10000
htru	17898	8	124959	2	news	39644	59	558454	1454
magic	19020	10	147097	2	qsar	546	8	1605	515
occupancy	20560	5	19721	2	query1	10000	3	29997	539
page	5473	10	9082	5					
raisin	900	7	6289	2					
rice	3810	7	24635	2					
room	10129	16	3072	4					
segment	2310	18	14910	7					
skin	245057	3	765	2					
wilt	4839	5	22599	2					

5.1. Experiment setup

All of our experiments were performed single-threaded on the DelftBlue supercomputer (Delft High Performance Computing Centre (DHPC) 2024) with Intel Xeon E5-6448Y processors and a memory limit of 8GB. Unless otherwise mentioned, we run the algorithm for a maximum of 30 minutes with a memory limit of 8GB of RAM. If either is exceeded, we report the best solution found at that point.

We list information on each dataset used in Table 5.1. We use the same datasets as in Mazumder et al. (2022), excluding the datasets that require a multidimensional prediction.[1]

The regularisation parameter we use for our method is $\lambda = \alpha x$ where x is a scalar depending on the dataset. x is $|\mathcal{D}|$ for classification and the loss of a single leaf node $l(\mathcal{D})$ for regression. Unless otherwise mentioned, we set $\alpha = 0$.

To test the significance of our results, we use the Friedman test combined with the post-hoc Nemenyi critical distance test with a significance level of 0.05, based on the guidelines of Demšar (2006). We perform these statistical significance tests with the autorank package (Herbold 2020).

5.2. Comparison to the state-of-the-art

We compare our method against two state-of-the-art methods for finding optimal decision trees with continuous features. ConTree (Brița et al. 2025), which finds optimal classification trees for arbitrary depth, and Quant-BnB (Mazumder et al. 2022), which finds optimal classification and regression trees for depths two and three.[2]

We compare the performance between these methods using two metrics. First, we use the runtime of the methods to evaluate the overall performance of the implementations. Second, we use the number of expansions of the (implicit) search graph to evaluate the search strategy independently of the efficiency of the implementation. We do not compare out-of-sample performance, as all of these methods find optimal trees.

Figure 5.1 shows the time taken to solve a number of datasets per method. For classification at depth two, only Quant-BnB takes more than a second to find the optimal decision tree for all datasets. There-

[1] Our experiment setup is available at `https://github.com/mimvdb/codt-experiments`.

[2] We use the reference implementations of these methods. For ConTree, we use the Python wrapper pycontree v1.0.4. For Quant-BnB, `https://github.com/mengxianglgal/Quant-BnB`, 0a381a5bc6d689f66e5fca34a767da2c21918da0.

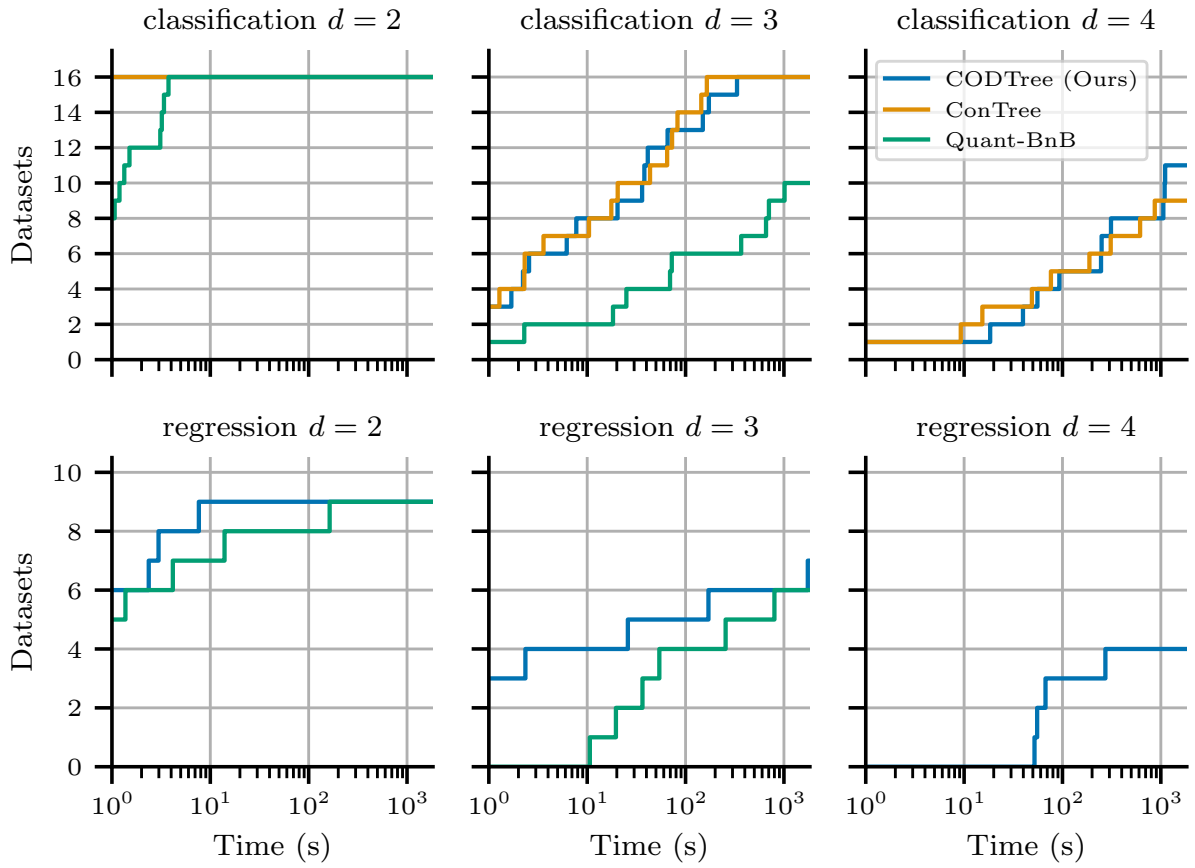


Figure 5.1: Number of datasets completed within some amount of time, per method.

fore, we focus on depths three and four. At these depths, our method is competitive with ConTree, and significantly faster than Quant-BnB. For regression, we are also significantly faster than Quant-BnB at all depths. To the best of our knowledge, we are the first to find optimal regression trees with continuous features at depth four.

Figure 5.2 compares the number of graph expansions of our method with ConTree, since they use a comparable search structure. In their context, the number of graph expansions is the number of general solver calls plus the number of specialised solver calls. We require fewer graph expansions than ConTree for all datasets. For depth two, this can be as extreme as a four-orders-of-magnitude difference. For larger depths, this difference is less extreme but still regularly an order of magnitude better.

Although we have significantly fewer graph expansions than ConTree, it has a comparable runtime. We argue that this is likely due to a more efficient implementation, not a better approach. Although our algorithm has some overhead due to the multiple implemented search strategies, around eighty percent of the overall time is spent in the left-right solver. We expect that the efficiency of our implementation of the left-right solver can be brought up to par with the equivalent solver present in ConTree.

5.3. Search strategy

To study what search strategy is best, we compare them by the time to prove optimality and by their anytime performance. We measure the anytime performance by saving intermediate upper and lower bounds after each expansion.

5.3.1. Time to optimality

We show the average rank of each search strategy in terms of time to optimality in Figure 5.3. This shows that global best-first search with the $h_{SmalltLB}$ heuristic, that selects the next feature test to ex-

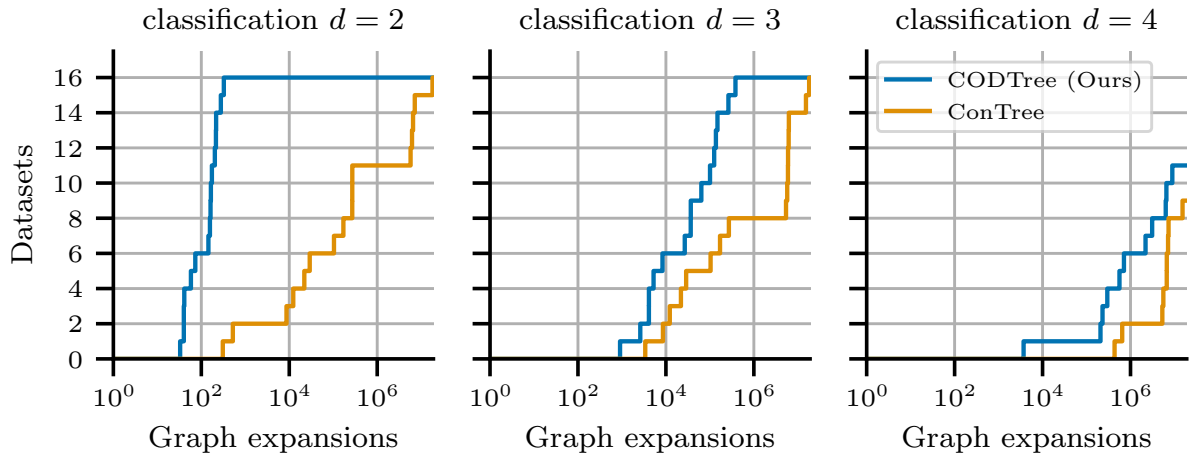


Figure 5.2: Number of datasets completed within some amount of search graph expansions, per method.

pand based on fewer instances remaining first and better lower bound second, performs best. Heuristics that do not prioritise smaller subproblems invariably run out of memory for $d > 2$. We found that the time to optimality correlates approximately linearly with the number of graph expansions for each problem instance, so we do not report these separately.

Figure 5.4 shows the magnitude of the difference between the search strategies that perform best. We do not include $h_{LB\&Small}$ as it is extremely similar to $h_{SmalltLB}$. The breakdown between classification and regression shows that the search strategies perform consistently well for classification and regression trees.

Easy problems While analysing the results, we noticed that the best search strategy differs significantly between easy and hard problem instances. We say that a dataset at a certain maximum depth is *easy* if all search strategies can find the optimal solution in ten seconds. When we only include easy problems, AND/OR search has the highest average rank, followed by h_{GOSDT}.

For easy problems, our depth-first search has a higher rank than the baseline depth-first search, but has a much lower rank than AND/OR search. For hard problems, our lower bound guided depth-first search seems worse than random depth-first search, but depth-first search in general performs relatively well.

The difference between easy and hard problems may explain the difference we see in our results compared to those from Chaouki et al. (2025), who suggest AND/OR search is better than depth-first and best-first search. Datasets with binary feature tests, and thus a much lower branching factor, might be more similar to our easy problems. The best search strategy for datasets with binary features remains an open question.

5.3.2. Anytime performance

To measure anytime performance, we integrate the best solution found over time. We do this by saving the intermediate upper and lower bounds after each graph expansion. We introduce the *objective integral* metric, inspired by the confined primal integral (Berthold 2013; Berthold and Csizmadia 2021) used for MIP solvers. It differs because, unlike MIP solvers, we can use the CART solution as an immediate upper bound on the loss. In addition, we are not concerned with the integral growing indefinitely for higher timeouts, as all compared methods tend to optimality.

We use the immediate upper bound to scale the metric for each dataset. We divide the integral by the objective value that CART would have achieved and by the maximum time taken by any method. We also subtract the lowest upper bound found, as this is shared between all methods. If any method finds the optimal solution, then the optimal solution is the lowest upper bound. This means that an objective integral of one is the worst, and an objective integral of zero is the best.

For the best found solution $u_i(t)$ and lower bound $l_i(t)$ at time t for method i, a timeout at time T, and

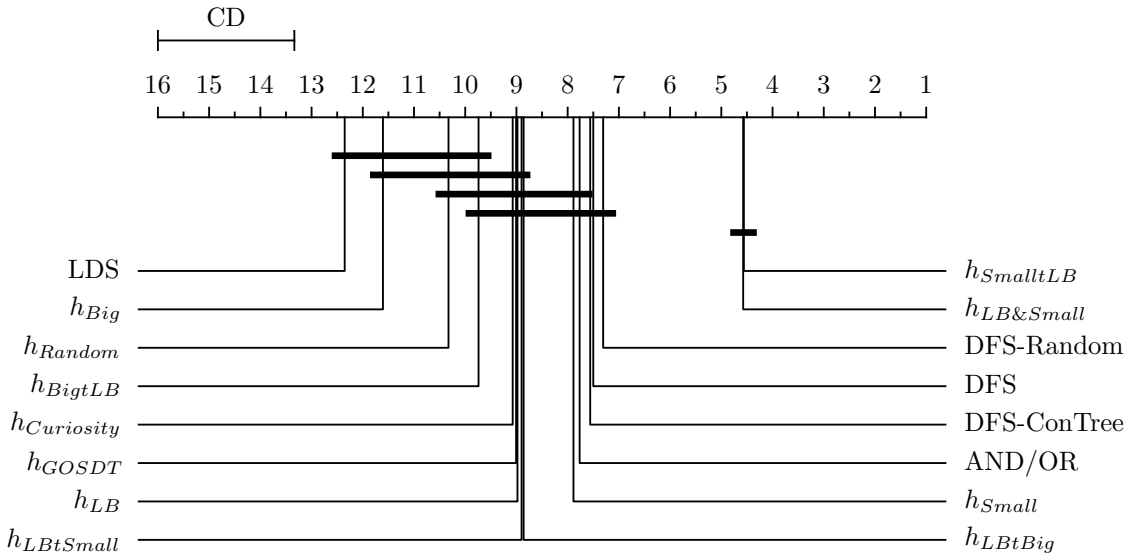


Figure 5.3: Mean rank of the runtime per search strategy. The critical distance diagram visualizes the results of the Nemenyi post-hoc significance test. The horizontal lines indicate groups where differences are not significant.

a CART solution U, the objective integral is

$$OI_i = \frac{1}{T(U - \min_j u_j(T))} \int_0^T (u_i(t) - \min_j u_j(T))dt \tag{5.1}$$

We use the *query1* dataset as an illustrative example to clarify the objective integral. Each plot in Figure 5.5 shows the bounds over time per search strategy. In this figure, U and $\min_j u_j(T)$ are shown as the upper and lower grey dashed lines respectively, while the upper blue and lower orange lines represent the upper and lower bound over time. The time to optimality is where the lower and upper bound become equal, and the objective integral is visually represented by the shaded blue area.

Figure 5.6 shows the average rank of each search strategy for the objective integral. In this figure, we exclude easy problem instances that all search strategies can solve in ten seconds, as we are interested in the anytime performance for problems that cannot be solved quickly. However, we note that there is no significant difference for the anytime performance when including or excluding easy problem instances.

We show the distribution of the objective integral for the three search strategies with the highest average rank in Figure 5.7 (excluding $h_{LB\&Small}$, as it is very similar to $h_{SmalltLB}$). Especially at depth four, $h_{SmalltLB}$ outperforms the other strategies. An interesting observation is that our depth-first search seems to have better anytime performance than the baseline depth-first search, but not better time to optimality.

LDS In our experiments, our limited discrepancy search has the lowest rank of all strategies in time to optimality. However, anytime performance, rather than time to optimality, is the main goal of this search strategy. But, while its average rank in the objective integral is much higher, it is not competitive with the other search strategies. This is in contrast to the results in Kiossou et al. (2022), where it outperforms depth-first search. There are several reasons why these results may differ. First, they use binary features while we have continuous features, which changes the way we rank feature tests (see Section 4.2.4). Second, our implementation of LDS may be lacking, as we use LDS in a best-first framework and without other prioritisation, causing it to reduce to h_{Random} when the number of discrepancies increases, rather than depth-first search as conventionally done (Kiossou et al. 2022).

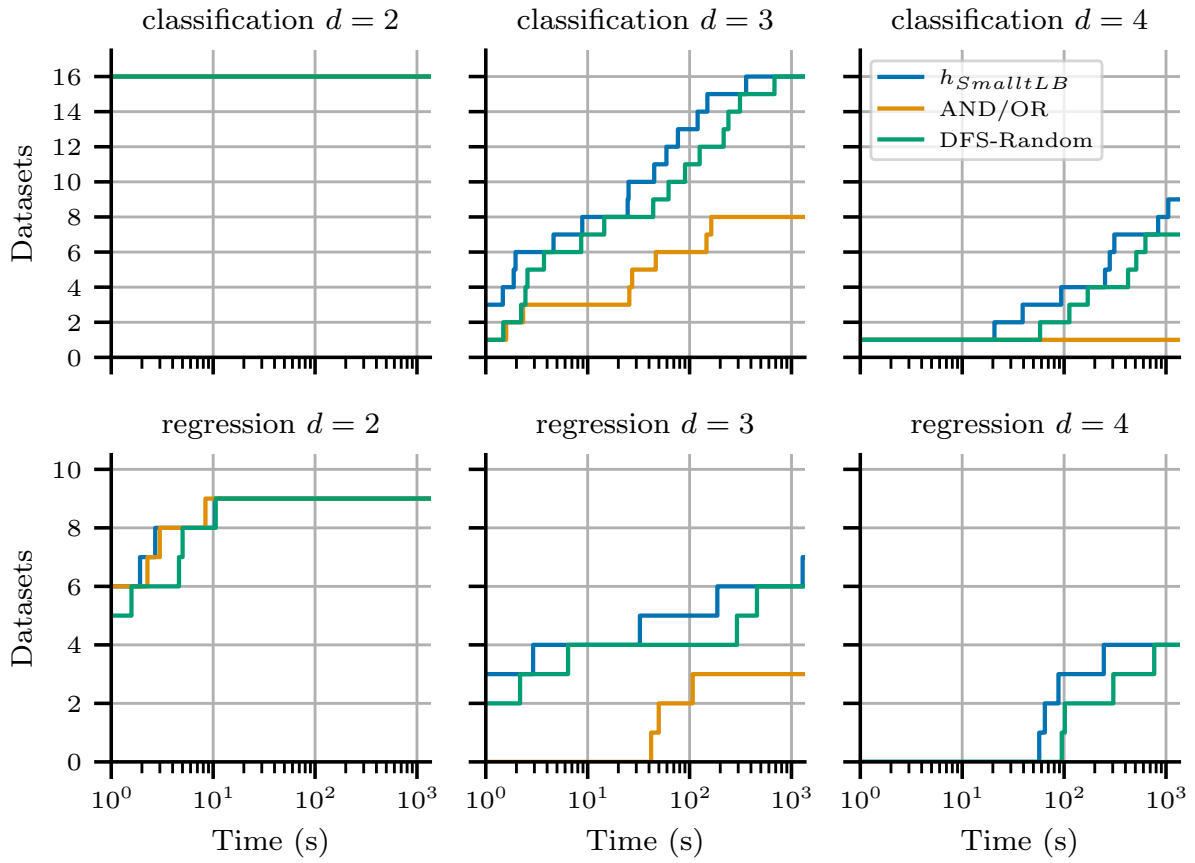


Figure 5.4: Number of datasets solved within some amount of time, per search strategy.

5.4. Ablation

The previous sections evaluated our algorithm in its entirety. In this section, we examine the contribution of two parts in particular. First, the contribution of the specialised solver for shallow trees. Second, the contribution of the clustering lower bound.

Specialised solvers for shallow trees We compare two specialised solvers for shallow trees from Section 4.4 in Figure 5.8. The first is the solver that solves the left and right sides of a depth-two tree simultaneously, the second solves depth-one trees exhaustively in $|\mathcal{F}|$ iterations over the data, the baseline does no exhaustive search and evaluates the loss of a leaf by iterating over the dataset.

The baseline is slowest by a large margin. This is expected as the dataset is explicitly split at each point before again iterating over the data to evaluate the loss. The depth one and the left-right solver are more comparable to each other. However, the left-right solver consistently takes less time as it does not need to explicitly construct the left and right nodes. Considering all datasets that can be solved with all three methods, the left-right solver and the depth-one solver have a geometric mean speedup over the baseline of 77.4x and 36.6x, respectively.

Clustering lower bound We test the impact of the clustering lower bound with the geometric mean difference of graph expansions with and without the lower bound, as well as the geometric mean difference in the time to optimality. For the datasets that find the optimal solution with and without the lower bound, including the lower bound reduces the number of graph expansions by two percent, while the time to optimality takes three percent longer. We do not see these differences as significant.

In Table 5.2, we show the fraction of the total optimal solution that the clustering lower bound is able to determine in the root, as well as the lower bound it can determine after a single feature test. This is generally higher than the regular clustering lower bound, but we cannot determine this bound efficiently

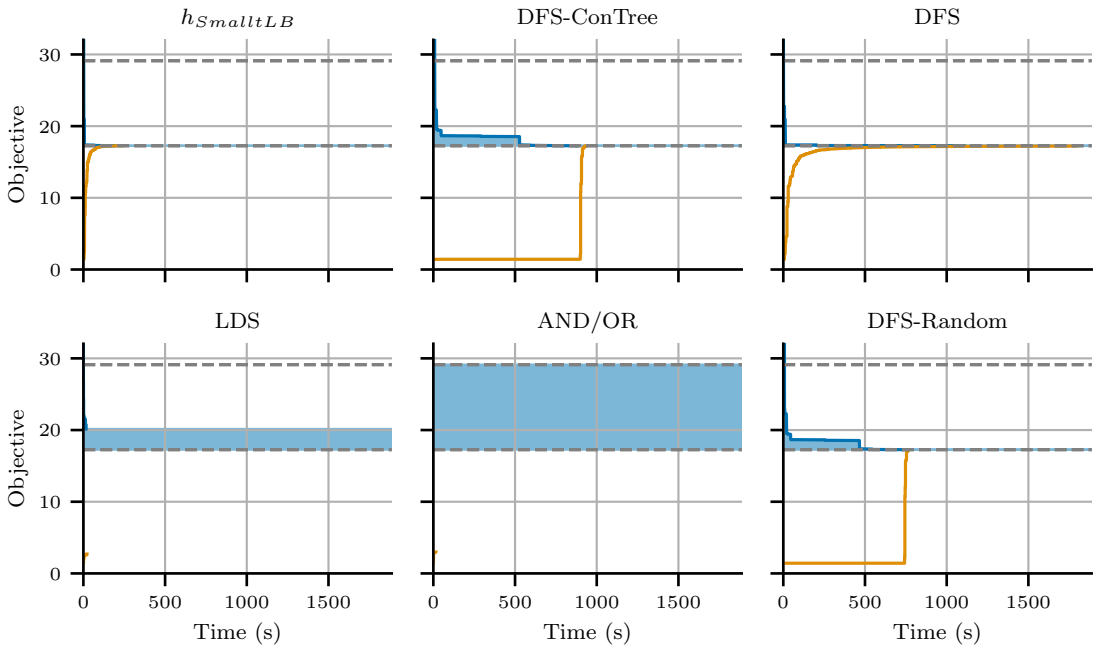


Figure 5.5: Upper (blue) and lower (orange) bounds over time for the query1 dataset ($d = 4$) for some search strategies. The grey lines indicate the training objective gap between CART and optimal. Interrupted lines indicate an out-of-memory error. The shaded blue area is the objective integral before scaling.

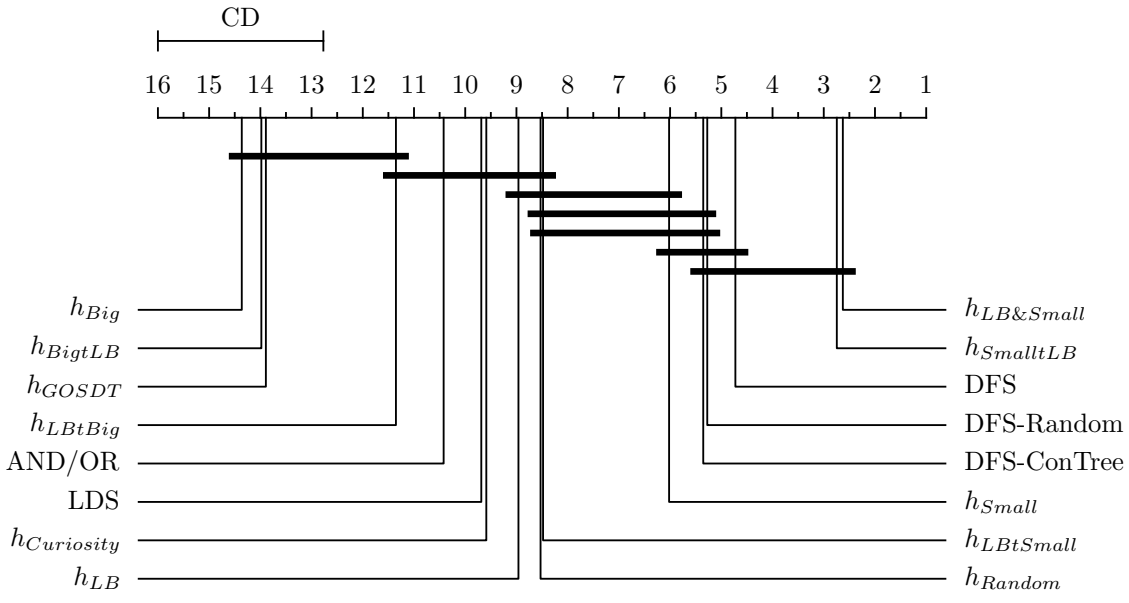


Figure 5.6: Mean rank of the objective integral per search strategy, excluding easy problem instances. The critical distance diagram visualizes the results of the Nemenyi post-hoc significance test. The horizontal lines indicate groups where differences are not significant.

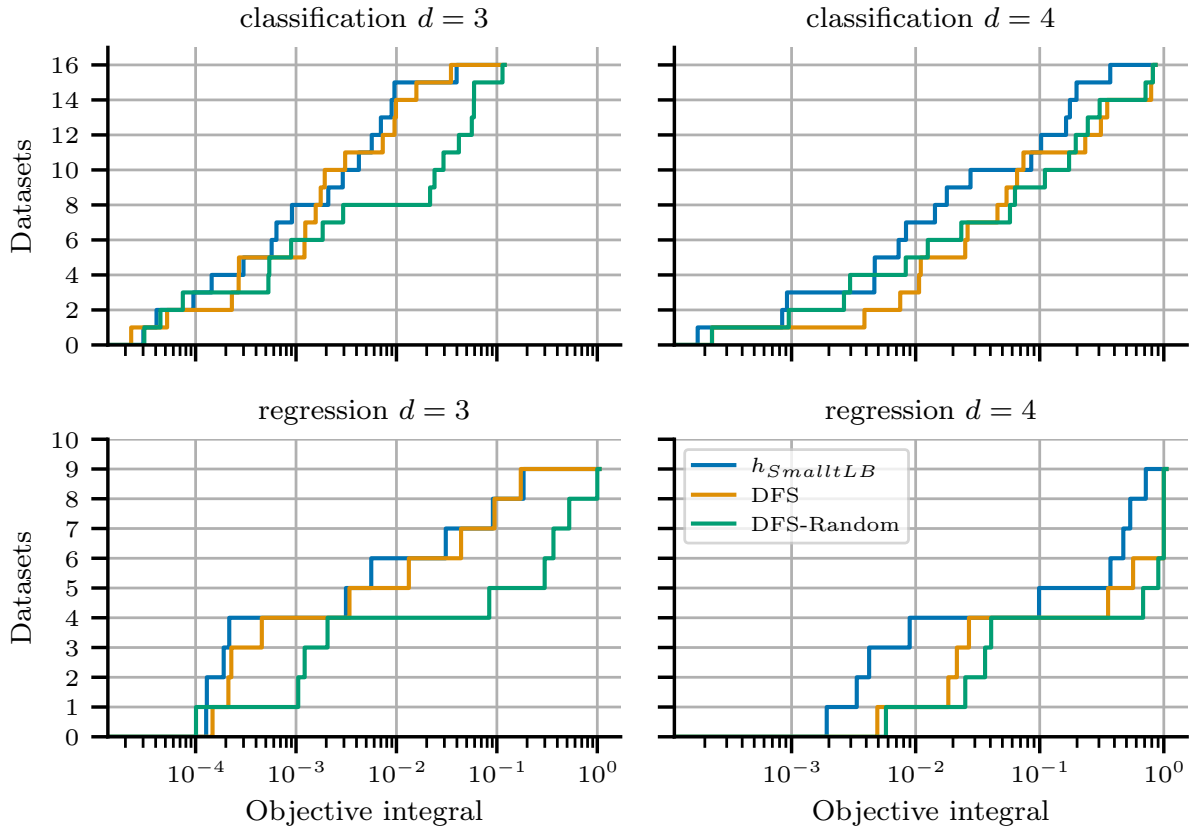


Figure 5.7: Distribution of the objective integral over the datasets, per search strategy.

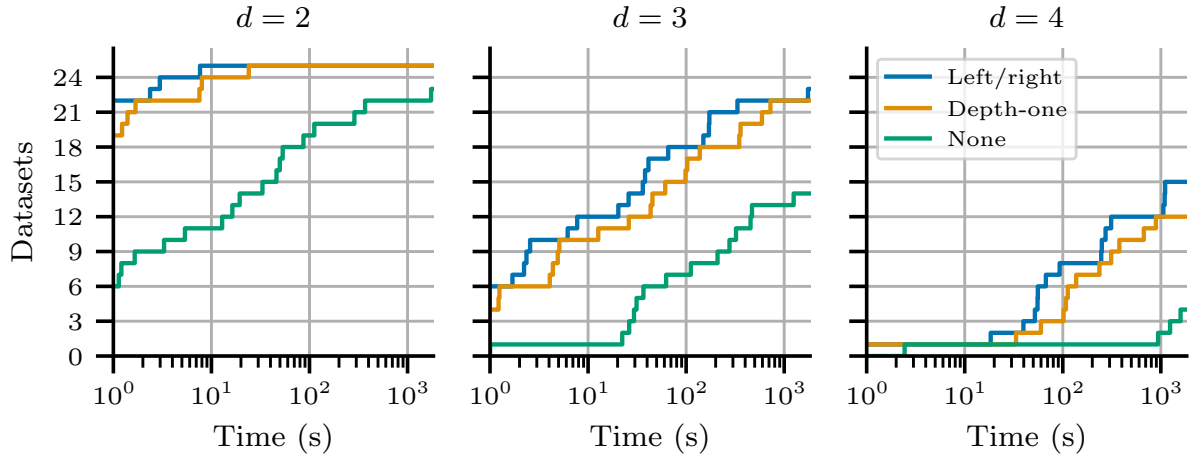


Figure 5.8: Number of datasets completed within some amount of time, per solver for shallow trees.

Table 5.2: The fraction of the optimal objective discovered as a lower bound by clustering, and by clustering after one feature test, for each dataset considering a maximum depth of three. For classification, we exclude binary classification datasets. Datasets marked with an asterisk (*) show the fraction of the best found solution instead of the optimal solution.

Dataset	Clustering	Depth-one clustering	Dataset	Clustering	Depth-one clustering
Classification			*Regression*		
avila	0.10	0.35	casp	0.02	0.06
bean	0.00	0.07	concrete	0.07	0.20
fault	0.00	0.17	energy*	0.03	0.09
page	0.00	0.00	fish	0.07	0.18
room	0.00	0.00	gas	0.67	0.72
segment	0.00	0.22	grid	0.04	0.12
			news*	0.05	0.15
			qsar	0.07	0.18
			query1	0.16	0.32

(see Section 4.5).

5.5. Generalisation of optimal decision trees

Previous experiments (Van der Linden et al. 2025) have shown that optimal *classification* trees are shallower and have a lower out-of-sample error than their greedy counterparts. Furthermore, the out-of-sample accuracy for optimal trees with continuous features is higher when using all continuous feature tests than when binning the data (Brița et al. 2025).

Following up on this, we perform out-of-sample experiments for *regression* trees. Zhang et al. (2023) claim that optimal gives higher out-of-sample accuracy than greedy, while Van den Bos et al. (2024) do not see a significant difference in the out-of-sample performance. However, the latter do not take into account the accuracy and interpretability trade-off. They only evaluate optimal decision trees with a maximum depth of five, and do not impose the same depth constraint on the greedy methods. Our experiments focus on shallow regression trees specifically, as Van der Linden et al. (2022) suggests that optimal decision trees perform especially well for shallow, interpretable trees.

We tune the regularisation parameter for our method and CART. [3]

For our method, we choose α from a fixed set, based on a nested five-fold cross-validation.

$$\alpha \in \{0.1, 0.05, 0.025, 0.01, 0.0075, 0.005, 0.0025, 0.001, 0.0005, 0.0001\} \tag{5.2}$$

We budget four hours for each dataset, one eleventh of the time used for each α divided equally over the cross validation, this is a little over four minutes each run. The remaining eleventh part, almost twenty-two minutes, is used to train on the full training data. If this time is insufficient, we report the best solution found.

For CART we tried two strategies for selecting the hyper parameters. The first is identical to our method. For the second, we chose thirty equi-log-spaced candidates from all possible $alpha$ parameters determined by the fully grown tree. This second approach results in better results more often, so we show these results.

For both methods, we use threshold tests of the form $f_i \leq x$ for all features.

We evaluate each method based on the coefficient of determination (R^2 score) on the test set. This is the proportion of the variance that the predictor explains. In our case, we can define the metric in terms of the SSE in a single leaf and the SSE for the tree $R^2(\tau, \mathcal{D}) = 1 - \frac{\mathcal{L}(\tau,\mathcal{D})}{l(\mathcal{D})}$, where $\mathcal{D}$ is the test set. We always use $\lambda = 0$ in the loss function to evaluate trees. The R^2 score is one for a perfect prediction, zero for a prediction that is as good as a constant prediction, and negative for anything that is worse.

[3] For CART, we use the implementation in scikit-learn v1.6.1.

Table 5.3: Out-of-sample R^2 results using five-fold cross validation. Best mean per depth in bold. $|\tau|$ is the number of branching nodes in the $d = 20$ CART tree. A * indicates that the method timed out before the optimal solution was found.

	$d = 2$		$d = 3$		$d = 4$		$d = 20$	
Dataset	CART	CODTree	CART	CODTree	CART	CODTree	CART	$\|\tau\|$
casp	**0.16**	**0.16**	**0.21**	**0.21***	**0.25**	0.22*	**0.48**	845
concrete	**0.48**	**0.48**	0.61	**0.63**	**0.70**	0.69*	**0.85**	673
energy	0.12	**0.13**	0.14	**0.15***	**0.16**	0.14*	**0.21**	2351
fish	**0.43**	0.40	0.44	**0.47**	**0.47**	**0.47***	**0.51**	24
gas	**0.94**	**0.94**	**0.97**	**0.97***	**0.98**	**0.98***	**1.00**	524
grid	0.19	**0.21**	0.30	**0.33***	0.40	**0.41***	**0.73**	276
news	**0.00**	**-0.00**	-0.09	**-0.00***	-0.19	**-0.00***	**-1.12**	3203
qsar	**0.30**	0.21	0.31	**0.43**	0.36	**0.43***	**0.35**	39
query1	0.77	**0.83**	0.88	**0.91**	0.91	**0.95***	**0.99**	5439

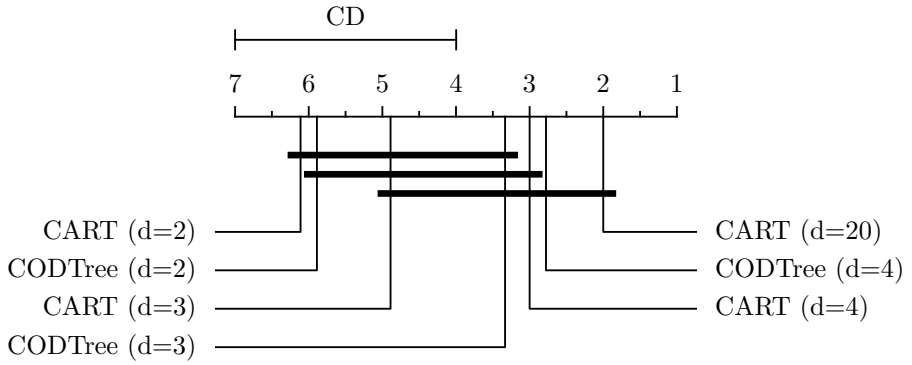


Figure 5.9: Mean rank per method of the out-of-sample R^2 score. The critical distance diagram visualizes the results of the Nemenyi post-hoc significance test. The horizontal lines indicate groups where differences are not significant.

The R^2 score on the test set for each method at depth two to four is shown in Table 5.3. For reference, we also show the R^2 test score for CART without a depth constraint. At all depths, optimal regression trees have a higher accuracy more often, and at depth three, the optimal regression tree is never worse than CART. However, as shown in Figure 5.9, we have insufficient data to reject the null hypothesis.

6

Conclusion

Finding optimal decision trees is a challenging problem for large datasets, and optimising for a large set of continuous feature tests drastically increases the branching factor and size of the search space. We introduce *CODTree*, an algorithm that can find optimal classification and regression trees for arbitrary depth. It is, to the best of our knowledge, the first to find optimal regression trees at depth four. Our algorithm can use different search strategies, and we show that choosing the right search strategy is essential for scalability.

Our experiments show that global best-first search with the $h_{SmalltLB}$ heuristic, which chooses the next feature test based on the remaining instances first and the best lower bound second, performs significantly better than other search strategies, a specialised procedure for shallow trees gives a geometric mean speedup of 77.4x, and that for our datasets optimal regression trees are never worse than CART at depth three, although we cannot conclude this last point to be significant.

We see three clear continuations for future work. First, our experiments determine the best search strategy for continuous features. However, it remains an open question whether this extends to binary datasets.

Second, our experiments did not conclusively show that optimal regression trees are better than CART. A more extensive experiment with more datasets may provide a clearer result. In addition, a more elaborate hypertuning procedure that leaves more time for the final training of the tree may be more effective, as many were not able to be trained to optimality due to the many repetitions required for cross-validation and parameter tuning.

Third, dynamic programming is very effective for increasing the scalability of datasets with binary features. However, in our preliminary experiments this is not the case for continuous features as there are many more distinct feature tests and repeats of the same sequence are rare. For this reason, we have not implemented it, but a search strategy that more regularly choses feature tests that have been chosen in the past, or an approximate caching approach that can take advantage of similar feature tests instead of only exact matches may provide a large improvement.

This thesis shows that it is feasible to find optimal decision trees for all continuous feature tests for at least depth four and that there are several promising paths that may improve this even further. This enables better interpretable models to be found than widely used greedy methods, leading to accuracy improvements and a reduction in the complexity of decision-making processes.

References

Aggarwal, Alok, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber (1987). “Geometric applications of a matrix-searching algorithm”. In: *Algorithmica* 2.1, pp. 195–208.

Aghaei, Sina, Mohammad Javad Azizi, and Phebe Vayanos (2019). “Learning Optimal and Fair Decision Trees for Non-Discriminative Decision-Making”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.1, pp. 1418–1426.

Aglin, Gaël, Siegfried Nijssen, and Pierre Schaus (2020). “Learning Optimal Decision Trees Using Caching Branch-and-Bound Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.4, pp. 3146–3153.

Aglin, Gaël, Siegfried Nijssen, and Pierre Schaus (2023). “Learning Optimal Decision Trees Under Memory Constraints”. In: *Machine Learning and Knowledge Discovery in Databases*. Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, Cham, pp. 393–409.

Ales, Zacharie, Valentine Huré, and Amélie Lambert (2024). “New optimization models for optimal classification trees”. In: *Computers & Operations Research* 164, p. 106515.

Amann, Julia, Alessandro Blasimme, Effy Vayena, Dietmar Frey, and Vince I. Madai (2020). “Explainability for artificial intelligence in healthcare: a multidisciplinary perspective”. In: *BMC Medical Informatics and Decision Making* 20.1, pp. 1–9.

Babbar, Varun, Hayden McTavish, Cynthia Rudin, and Margo Seltzer (2025). “Near-Optimal Decision Trees in a SPLIT Second”. In: Forty-second International Conference on Machine Learning.

Barredo Arrieta, Alejandro et al. (2020). “Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI”. In: *Information Fusion* 58, pp. 82–115.

Bellman, Richard (1973). “A note on cluster analysis and dynamic programming”. In: *Mathematical Biosciences* 18.3, pp. 311–312.

Berthold, Timo (2013). “Measuring the impact of primal heuristics”. In: *Operations Research Letters* 41.6, pp. 611–614.

Berthold, Timo and Zsolt Csizmadia (2021). “The confined primal integral: a measure to benchmark heuristic MINLP solvers against global MINLP solvers”. In: *Mathematical Programming* 188.2, pp. 523–537.

Bertsimas, Dimitris and Jack Dunn (2017). “Optimal classification trees”. In: *Machine Learning* 106.7, pp. 1039–1082.

Bertsimas, Dimitris, Jack Dunn, and Aris Paschalidis (2017). “Regression and classification using optimal decision trees”. In: *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pp. 1–4.

Biggs, David, Barry De Ville, and Ed Suen (1991). “A method of choosing multiway partitions for classification and decision trees”. In: *Journal of Applied Statistics* 18.1, pp. 49–62.

Blanc, Guy, Jane Lange, Chirag Pabbaraju, Colin Sullivan, Li-Yang Tan, and Mo Tiwari (2023). “Harnessing the power of choices in decision tree learning”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pp. 80220–80232.

Blanquero, Rafael, Emilio Carrizosa, Cristina Molero-Río, and Dolores Romero Morales (2021). “Optimal randomized classification trees”. In: *Computers & Operations Research* 132, p. 105281.

Van den Bos, Mim, Jacobus G. M. van der Linden, and Emir Demirović (2024). “Piecewise Constant and Linear Regression Trees: An Optimal Dynamic Programming Approach”. In: *Proceedings of the 41st International Conference on Machine Learning*, pp. 48994–49007.

Breiman, Leo, Jerome Friedman, Charles J. Stone, and R. A. Olshen (1984). *Classification and Regression Trees*. Taylor & Francis.

Briță, Cătălin E., Jacobus G. M. van der Linden, and Emir Demirović (2025). “Optimal Classification Trees for Continuous Feature Data Using Dynamic Programming with Branch-and-Bound”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 39.11, pp. 11131–11139.

Carrizosa, Emilio, Cristina Molero-Río, and Dolores Romero Morales (2021). “Mathematical optimization in classification and regression trees”. In: *TOP* 29.1, pp. 5–33.

Chaouki, Ayman, Jesse Read, and Albert Bifet (2025). “Branches: Efficiently Seeking Optimal Sparse Decision Trees via AO*”. In: Forty-second International Conference on Machine Learning.

Costa, Vinícius G. and Carlos E. Pedreira (2023). “Recent advances in decision trees: an updated survey”. In: *Artificial Intelligence Review* 56.5, pp. 4765–4800.

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

Demirović, Emir, Emmanuel Hebrard, and Louis Jean (2023). “Blossom: an Anytime Algorithm for Computing Optimal Decision Trees”. In: *Proceedings of the 40th International Conference on Machine Learning*, pp. 7533–7562.

Demirović, Emir, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey (2022). “MurTree: optimal decision trees via Dynamic programming and search”. In: *Journal of Machine Learning Research* 23.26, pp. 1–47.

Demšar, Janez (2006). “Statistical Comparisons of Classifiers over Multiple Data Sets”. In: *Journal of Machine Learning Research* 7.1, pp. 1–30.

Du, Mengnan, Ninghao Liu, and Xia Hu (2019). “Techniques for interpretable machine learning”. In: *Communications of the ACM* 63.1, pp. 68–77.

Dunn, Jack (2018). “Optimal trees for prediction and prescription”. Thesis. Massachusetts Institute of Technology.

Fenwick, Peter M. (1994). “A new data structure for cumulative frequency tables”. In: *Software: Practice and Experience* 24.3, pp. 327–336.

Garey, M. R. and R. L. Graham (1974). “Performance bounds on the splitting algorithm for binary testing”. In: *Acta Informatica* 3.4, pp. 347–355.

Garrett, Brandon L. and Cynthia Rudin (2023). “Interpretable algorithmic forensics”. In: *Proceedings of the National Academy of Sciences* 120.41.

Günlük, O., J. Kalagnanam, M. Li, M. Menickelly, and K. Scheinberg (2021). “Optimal decision trees for categorical data via integer programming”. In: *Journal of Global Optimization* 81.1, pp. 233–260.

Herbold, Steffen (2020). “Autorank: A Python package for automated ranking of classifiers”. In: *Journal of Open Source Software* 5.48, p. 2173.

Hu, Xiyang, Cynthia Rudin, and Margo Seltzer (2019). “Optimal Sparse Decision Trees”. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Vol. 32, pp. 7267–7275.

Hua, Kaixun, Jiayang Ren, and Yankai Cao (2022). “A Scalable Deterministic Global Optimization Algorithm for Training Optimal Decision Tree”. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. Vol. 35, pp. 8347–8359.

Hyafil, Laurent and Ronald L. Rivest (1976). “Constructing optimal binary decision trees is NP-complete”. In: *Information Processing Letters* 5.1, pp. 15–17.

Janota, Mikoláš and António Morgado (2020). “SAT-Based Encodings for Optimal Decision Trees with Explicit Paths”. In: *Theory and Applications of Satisfiability Testing – SAT 2020*, pp. 501–518.

Jo, Nathanael, Sina Aghaei, Jack Benson, Andres Gomez, and Phebe Vayanos (2023). “Learning Optimal Fair Decision Trees: Trade-offs Between Interpretability, Fairness, and Accuracy”. In: *Proceedings of the 2023 AAAI/ACM Conference on AI, Ethics, and Society*, pp. 181–192.

Justin, Nathan, Sina Aghaei, Andres Gomez, and Phebe Vayanos (2021). “Optimal Robust Classification Trees”. In: The AAAI-22 Workshop on Adversarial Machine Learning and Beyond.

Kass, G. V. (1980). “An Exploratory Technique for Investigating Large Quantities of Categorical Data”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 29.2, pp. 119–127.

Kiossou, Harold, Pierre Schaus, Siegfried Nijssen, and Gaël Aglin (2024). “Efficient Lookahead Decision Trees”. In: *Advances in Intelligent Data Analysis XXII*, pp. 133–144.

Kiossou, Harold, Pierre Schaus, Siegfried Nijssen, and Vinasetan Ratheil Houndji (2022). “Time Constrained DL8.5 Using Limited Discrepancy Search”. In: *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 443–459.

Lin, Jimmy, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer (2020). “Generalized and Scalable Optimal Sparse Decision Trees”. In: *Proceedings of the 37th International Conference on Machine Learning*, pp. 6150–6160.

Van der Linden, Jacobus, Mathijs de Weerdt, and Emir Demirović (2022). “Fair and Optimal Decision Trees: A Dynamic Programming Approach”. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems* 35, pp. 38899–38911.

Van der Linden, Jacobus G. M., Daniël Vos, Mathijs M. de Weerdt, Sicco Verwer, and Emir Demirović (2025). *Optimal or Greedy Decision Trees? Revisiting their Objectives, Tuning, and Performance*. arXiv preprint arXiv:2409.12788.

Van der Linden, Jacobus G. M., Mathijs de Weerdt, and Emir Demirović (2023). “Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming”. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Vol. 36, pp. 9173–9212.

Loh, Wei-Yin (2002). “Regression trees with unbiased variable selection and interaction detection”. In: *Statistica Sinica* 12, pp. 361–386.

Martelli, Alberto and Ugo Montanari (1978). “Optimizing decision trees through heuristically guided search”. In: *Communications of the ACM* 21.12, pp. 1025–1039.

Mazumder, Rahul, Xiang Meng, and Haoyue Wang (2022). “Quant-BnB: A Scalable Branch-and-Bound Method for Optimal Decision Trees with Continuous Features”. In: *Proceedings of the 39th International Conference on Machine Learning*, pp. 15255–15277.

McTavish, Hayden, Chudi Zhong, Reto Achermann, Ilias Karimalis, Jacques Chen, Cynthia Rudin, and Margo Seltzer (2022). “Fast Sparse Decision Tree Optimization via Reference Ensembles”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.9. Number: 9, pp. 9604–9613.

Moret, Bernard M. E. (1982). “Decision Trees and Diagrams”. In: *ACM Computing Surveys* 14.4, pp. 593–623.

Morgan, James N. and John A. Sonquist (1963). “Problems in the Analysis of Survey Data, and a Proposal”. In: *Journal of the American Statistical Association* 58.302, pp. 415–434.

Narodytska, Nina, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva (2018). “Learning Optimal Decision Trees with SAT”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pp. 1362–1368.

Nijssen, Siegfried and Elisa Fromont (2007). “Mining optimal decision trees from itemset lattices”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 530–539.

Nilsson, N. J. (1980). *Principles of artificial intelligence*. Reprinted by Morgan Kaufmann (2014). Palo Alto, CA.: Tioga Publishing Co.

Norouzi, Mohammad, Maxwell Collins, Matthew A Johnson, David J Fleet, and Pushmeet Kohli (2015). “Efficient Non-greedy Optimization of Decision Trees”. In: *Proceedings of the 29th International Conference on Neural Information Processing Systems*. Vol. 28, pp. 1729–1737.

Ordyniak, Sebastian and Stefan Szeider (2021). “Parameterized Complexity of Small Decision Tree Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.7, pp. 6454–6462.

Quinlan, J. R. (1986). “Induction of decision trees”. In: *Machine Learning* 1.1, pp. 81–106.

Quinlan, J. Ross (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.

Reinwald, Lewis T. and Richard M. Soland (1966). “Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time”. In: *Journal of the ACM* 13.3, pp. 339–358.

Rudin, Cynthia (2019). “Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead”. In: *Nature Machine Intelligence* 1.5, pp. 206–215.

Rudin, Cynthia, Chaofan Chen, Zhi Chen, Haiyang Huang, Lesia Semenova, and Chudi Zhong (2022). “Interpretable machine learning: Fundamental principles and 10 grand challenges”. In: *Statistics Surveys* 16, pp. 1–85.

Shati, Pouya, Eldan Cohen, and Sheila A. McIlraith (2023). “SAT-based optimal classification trees for non-binary data”. In: *Constraints* 28.2, pp. 166–202.

Song, Mingzhou and Hua Zhong (2020). “Efficient weighted univariate clustering maps outstanding dysregulated genomic zones in human cancers”. In: *Bioinformatics* 36.20, pp. 5027–5036.

Staus, Luca Pascal, Christian Komusiewicz, Frank Sommer, and Manuel Sorge (2025). “Witty: An Efficient Solver for Computing Minimum-Size Decision Trees”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 39.19, pp. 20584–20591.

Sullivan, Colin, Mo Tiwari, and Sebastian Thrun (2024). “MAPTree: Beating “Optimal” Decision Trees with Bayesian Decision Trees”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.8, pp. 9019–9026.

Verhaeghe, Hélène, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus (2020). "Learning optimal decision trees using constraint programming". In: *Constraints* 25.3, pp. 226–250.

Verwer, Sicco and Yingqian Zhang (2019). "Learning Optimal Classification Trees Using a Binary Linear Program Formulation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.1, pp. 1625–1632.

Vos, Daniël and Sicco Verwer (2022). "Robust Optimal Classification Trees against Adversarial Examples". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.8, pp. 8520–8528.

Wang, Haizhou and Mingzhou Song (2011). "Ckmeans.1d.dp: Optimal k-means Clustering in One Dimension by Dynamic Programming". In: *The R Journal* 3.2, pp. 29–33.

Xue, S., X. Luo, and C. Gao (2024). "Nonlinear Optimal Classification Trees". In: 2024 5th International Conference on Machine Learning and Computer Application, pp. 635–639.

Zhang, R., R. Xin, M. Seltzer, and C. Rudin (2023). "Optimal Sparse Regression Trees". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37, pp. 11270–11279.

A

Derivations

In several of our claims, we make use of a decomposition of the regression loss for a single leaf into the sum, the sum of squares, and the sample size. This is not a novel result, but we provide the derivation below for clarity.

Proof. Let τ be a tree that is a single leaf node. And define $\bar{y} = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} y$.

$$\mathcal{L}_r(\tau, \mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} (y - \tau(x))^2 \tag{A.1}$$

$$= \sum_{(x,y)\in\mathcal{D}} (y - \bar{y})^2 \qquad \text{(since } \tau \text{ is a leaf)} \tag{A.2}$$

$$= \sum_{(x,y)\in\mathcal{D}} (y^2 - 2y\bar{y} + \bar{y}^2) \qquad \text{(expand)} \tag{A.3}$$

$$= \sum_{(x,y)\in\mathcal{D}} y^2 - 2|\mathcal{D}|\bar{y}^2 + |\mathcal{D}|\bar{y}^2 \qquad \text{(distribute sum and simplify)} \tag{A.4}$$

$$= \sum_{(x,y)\in\mathcal{D}} y^2 - \frac{\left(\sum_{(x,y)\in\mathcal{D}} y\right)^2}{|\mathcal{D}|} \qquad \square$$