

Anytime Minimum-Size Perfect Decision Trees

Generation of Perfect Trees with Continuous
Features

Thomas Theodor Corné Rooskens

Delft University of Technology

Anytime Minimum-Size Perfect Decision Trees

Generation of Perfect Trees with Continuous
Features

by

Thomas Theodor Corné Rooskens

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Monday October 12, 2026 at 01:30 PM.

Student number:	5554950	
Project duration:	November 17, 2025 – July 15, 2026	
Thesis committee:	Dr. Emir Demirović,	TU Delft, supervisor
	Dr.ir. Jacobus G. M. van der Linden,	TU Delft, daily supervisor
	Dr. Jérémie Decouchant,	TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Contents

1	Introduction	2
2	Related Work	4
2.1	Optimal Decision Trees	4
2.1.1	Accuracy	4
2.1.2	Size	5
2.2	CODTree	6
2.2.1	Search Strategies	7
2.3	Research Gap	7
3	Preliminaries	9
3.1	Notation and Problem Definition	9
3.2	CODTree	10
3.3	Set Cover as a Lower Bound	11
3.3.1	Solving the Set Cover as a Linear Program	13
3.3.2	A Simple Approach for the Set Cover Problem	13
3.4	Sub-interval Pruning	14
4	Methodology	15
4.1	Finding Perfect Trees	15
4.2	Bounds	15
4.2.1	Class-count Lower Bound	15
4.2.2	One-off Lower Bound	16
4.2.3	Pair and Pair-size Lower Bound	16
4.2.4	CART Upper Bound and Anytime Performance	16
4.3	Memory Aware Search Strategy	17
4.4	Data Reduction	18
5	Experiments	19
5.1	Experiment Setup	19
5.2	Comparing Search Strategies	19
5.3	Comparison with a State-of-the-Art Method	21
5.4	Effectiveness of the Bounds	23
6	Conclusion	25
	References	27
A	Datasets	29
B	Search Strategies Performance	30
C	Similarity Lower Bound	36

Abstract

This thesis addresses the Minimum-Size Decision Tree (MSDT) problem, which seeks the smallest decision tree that perfectly classifies all training data. Smaller perfect decision trees may generalize better to unseen data, are easier to interpret and validate, require less memory, and have faster inference. These properties are particularly valuable in high-stakes domains such as healthcare (Amann et al. 2020), where decisions must be understandable and auditable, and in resource-constrained systems such as microcontrollers, where memory and computation power are limited. Existing methods for continuous features mainly optimize classification accuracy under a depth constraint, while specialized MSDT methods do not exploit branch-and-bound techniques developed for continuous features and may fail to return a valid perfect tree when the search is interrupted. To address this gap, we extend a branch-and-bound framework for continuous features to the MSDT setting by redefining the objective and adding several problem-specific lower bounds alongside a randomized CART-based upper bound. Additionally, we propose a memory-aware hybrid search strategy to balance efficiency and resource usage. Experimental results show that the proposed approach provides useful anytime performance by maintaining a valid perfect decision tree throughout the search and progressively improving its size as the search continues. Although it proves optimality on fewer difficult instances than a state-of-the-art baseline, it can still return increasingly strong feasible solutions when the search is interrupted. The experiments further show that the pair-size lower bound is the most effective practical bound, while the memory-aware hybrid search strategy reduces the risk of memory exhaustion during search.

1

Introduction

Decision trees are a class of machine learning models that are widely adopted due to their interpretability, which is becoming increasingly important in domains such as healthcare (Amann et al. 2020) and criminal justice (Garrett and Rudin 2023). An example of such a decision tree is shown in Figure 1.1.

Traditionally, decision trees have been constructed by heuristic algorithms such as CART (Breiman et al. 1984) and C4.5 (Salzberg 1994). While these algorithms are fast and often produce well-performing decision trees, they do not guarantee that the resulting tree is *optimal*. In other words, the constructed tree may not be the best possible representation of the data with respect to accuracy, size, or another objective.

An alternative to these heuristic approaches is to construct *optimal decision trees*, i.e., decision trees that are optimal according to a given objective function. However, constructing such trees is NP-hard (Hyafil and Rivest 1976; Ordyniak and Szeider 2021), making the problem computationally demanding in practice.

In this thesis, we consider the size objective, also known as the Minimum-Size Decision Tree (MSDT) problem, where size is measured by the number of nodes in the tree. The goal of MSDT is to find the smallest decision tree that perfectly classifies all training examples. Such trees are of interest because smaller decision trees may generalize better to out-of-sample data (Lin et al. 2020), are easier for domain experts to inspect and validate, require less memory, and have lower inference latency than larger trees. These properties are particularly valuable in applications such as medical diagnosis and credit assessment, where predictions must be understandable and auditable, and in resource-constrained embedded systems, such as microcontrollers and edge devices, where memory, computing power, and response time are limited.

In recent years, substantial algorithmic progress has been made on the construction of optimal decision trees for accuracy optimization. Approaches based on dynamic programming (Aglin, Nijssen, and Schaus 2020; Lin et al. 2020; Demirović et al. 2022) can compute provably optimal trees for a given dataset. These systems typically rely on pruning strategies, caching schemes, and bounding functions to improve efficiency while maintaining formal optimality guarantees.

Existing methods that directly target minimum-size perfect decision trees based on witness trees (Staus et al. 2025) and Boolean satisfiability (SAT) (Narodytska et al. 2018; Avellaneda 2020; Janota and Morgado 2020) have made substantial progress, but they remain limited by pruning efficiency and by the absence of anytime feasible solutions. In particular, if the search is interrupted before optimality is proven, such methods may not return a valid perfect decision tree.

To address these challenges, this work builds on a modular algorithm for computing optimal decision trees with continuous features, which supports different search strategies (van den Bos 2025). We choose this algorithm as the basis of our approach because its modular search framework allows the search strategy, objective function, and pruning mechanisms to be adapted without redesigning the

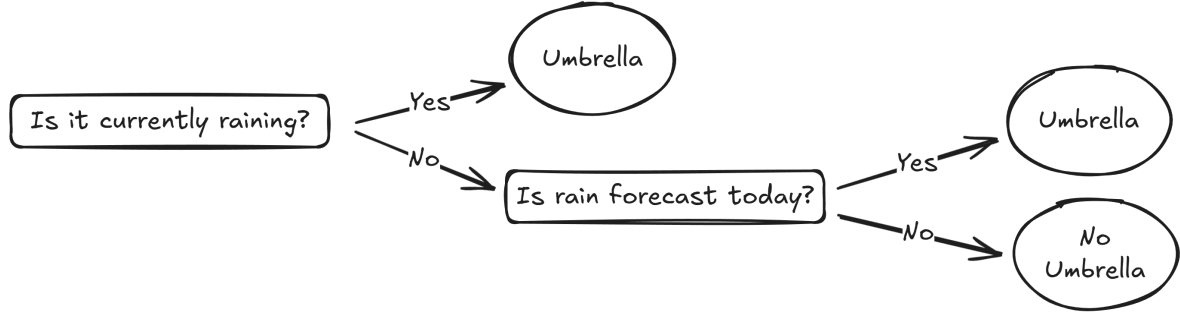


Figure 1.1: A decision tree for deciding whether to bring an umbrella.

complete optimization procedure. Moreover, the algorithm already includes methods specifically designed to handle continuous features, which further supports its suitability for addressing the MSDT problem on such datasets. The proposed extension removes the depth constraint, restricts the search to pure leaf nodes, and redefines the objective to minimize tree size. To improve search efficiency, we apply several complementary lower bounds, together with a randomized CART-based upper bound that provides strong initial solutions and enhances anytime performance. Additionally, we propose a memory-aware hybrid search strategy to balance exploration quality and resource usage during search.

Experiments on benchmark datasets show that the proposed method performs well on smaller instances and provides useful anytime behavior. While it solves fewer datasets overall than one of the state-of-the-art algorithms by (Staus et al. 2025), it often requires fewer search graph expansions and consistently returns a valid solution when interrupted.

In summary, the main contributions of this work are:

- an extension of an optimal decision tree algorithm to the MSDT problem;
- the integration of multiple lower-bound techniques to improve pruning efficiency;
- a randomized CART-based upper bound that improves anytime feasible solutions;
- a memory-aware hybrid search strategy that balances search efficiency and memory usage.

The remainder of this thesis is structured as follows. Chapter 2 reviews the necessary background and related work. Chapter 3 introduces the notation, problem definition, CODTree framework, and lower-bound concepts used throughout the thesis. Chapter 4 presents the proposed approach and its algorithmic components. Chapter 5 reports and analyzes the experimental results. Chapter 6 concludes with a discussion of limitations and directions for future research.

2

Related Work

This chapter reviews the work most relevant to this thesis. We first discuss methods for constructing optimal decision trees, including dynamic programming approaches and branch-and-bound algorithms for continuous features. We then review methods for constructing minimum-size perfect decision trees, including SAT-based formulations and the witness-tree-based Witty algorithm. Next, we describe CODTree, the branch-and-bound framework extended in this thesis, together with the search strategies it supports. Finally, we identify the research gap addressed by this work.

2.1. Optimal Decision Trees

An *optimal decision tree* is a decision tree that is optimal with respect to a specified objective function on the training data. Depending on the setting, this objective may involve minimizing classification error or minimizing the size or depth of the tree. However, finding such a tree is NP-hard because there are many possible tree structures, features, and split points to consider, and the number of combinations grows rapidly as the dataset becomes larger (Ordyniak and Szeider 2021). In this section, we will provide an overview of different methods for building optimal decision trees.

We first review methods for computing decision trees that are optimal with respect to accuracy, followed by methods that optimize tree size.

2.1.1. Accuracy

Dynamic Programming

Dynamic programming is a commonly used approach for finding optimal decision trees. It exploits the fact that, once a split is selected, the resulting subtrees can be optimized independently. The original problem can therefore be decomposed into smaller subproblems corresponding to subsets of the training data. By storing the optimal solutions to these subproblems, the algorithm avoids solving the same subproblem multiple times.

Nijssen and Fromont (2007) introduced DL8, one of the first algorithms for constructing optimal decision trees. The algorithm represents each subproblem by the branching decisions that led to this subproblem. When the same subset is encountered again, the previously computed solution can be reused, avoiding redundant computation and reducing the size of the search space.

Building on this work, Aglin, Nijssen, and Schaus (2020) developed DL8.5, which extends DL8 with branch-and-bound pruning and a stronger caching scheme. Rather than storing only solved subproblems, DL8.5 also caches subproblems that have been pruned. As a result, the algorithm avoids revisiting regions of the search space that cannot improve the current best solution.

Demirović et al. (2022) further improved on DL8.5 with MurTree. A key contribution is a specialized procedure for constructing trees of depth at most two, which computes optimal shallow trees more efficiently by considering feature pairs directly. In addition, MurTree introduces a similarity-based lower bound that reuses lower bounds computed for previously encountered, similar datasets.

Rather than focusing solely on improving search efficiency, van der Linden, de Weerd, and Demirović (2023) extended the applicability of dynamic programming with *STreeD*. Whereas earlier methods optimize additive objectives such as classification error, *STreeD* supports any separable objective that satisfies the required mathematical conditions. This enables the construction of optimal decision trees for objectives including fairness and nonlinear classification metrics.

These dynamic programming methods show the importance of caching, lower bounds, and pruning for decision tree construction. However, they are mainly designed for depth-constrained optimal decision trees, where the goal is to minimize an objective such as classification error. In this thesis, the goal is different: we search for the smallest decision tree that perfectly classifies the training data.

Continuous Features

Many decision tree algorithms assume that all features are binary. Although continuous features can be transformed into binary features by introducing a threshold for each possible split, this transformation does not exploit the fact that these binary features originate from the same continuous feature. Several methods have therefore been proposed to handle continuous features directly.

Lin et al. (2020) proposed *GOSDT*, which extends dynamic programming methods to support continuous features and a wider range of objective functions. Instead of optimizing only the classification error, *GOSDT* supports objectives such as balanced accuracy, F_1 -score, and AUC. To efficiently handle continuous features, the algorithm represents similar subproblems using shared subtrees.

Rather than extending dynamic programming, Mazumder, Meng, and Wang (2022) proposed *Quant-BnB*, a branch-and-bound algorithm for datasets with continuous features. The algorithm partitions the possible split thresholds into intervals based on feature quantiles and computes lower and upper bounds for each interval. These bounds allow large parts of the search space to be pruned without evaluating every possible threshold individually. However, the authors report that the approach is mainly effective for shallow trees of depth two or three.

More recently, Brita, van der Linden, and Demirović (2025) introduced *ConTree*, a branch-and-bound algorithm designed specifically for continuous features. *ConTree* introduces several pruning techniques that exploit the ordering of continuous feature values, including sub-interval pruning. In addition, the algorithm uses a specialized procedure for trees of depth two that evaluates all splits efficiently. Together, these techniques significantly reduce the number of candidate thresholds that need to be explored, making *ConTree* the first method capable of constructing optimal decision trees of depth four on continuous datasets within practical time limits.

CODTree (van den Bos 2025) generalizes these developments into a modular branch-and-bound framework for optimal decision trees with continuous features. Since *CODTree* forms the basis of this thesis, we discuss the framework and its search strategies in more detail in section 2.2.

2.1.2. Size

Unlike optimal decision trees where the objective is to minimize the classification error subject to a depth constraint, we will now consider the construction of an optimal tree that correctly classifies every training example, also called a perfect decision tree. Among all perfect decision trees, we are interested in finding the one with the smallest number of decision nodes. This problem is known as the Minimum-Size Decision Tree (MSDT) problem and is the focus of this thesis.

SAT

An alternative approach for constructing decision trees is to formulate the problem as a Boolean satisfiability (SAT) problem. Instead of recursively exploring the search space, these methods encode the existence of a decision tree satisfying a set of constraints as a propositional formula. A SAT solver is then used to determine whether such a tree exists.

Narodytska et al. (2018) proposed one of the first SAT formulations for learning optimal decision trees. Their encoding represents the structure of the decision tree, the selected feature at each internal node, the class assigned to each leaf, and the routing of training examples using Boolean variables. Given a maximum tree depth, the SAT solver determines whether a decision tree satisfying all constraints exists.

Several improvements have since been proposed to improve the scalability of SAT-based methods. Avellaneda (2020) introduced an incremental formulation that only adds training examples that invalidate the current candidate tree, significantly reducing the size of the SAT instances. Janota and Morgado (2020) instead encode complete root-to-leaf paths rather than individual tree nodes, resulting in a more compact encoding. More recently, Shati, Cohen, and McIlraith (2023) extended SAT formulations to support continuous features directly by encoding threshold comparisons without requiring a preprocessing step.

SAT-based methods have demonstrated competitive performance for this problem. However, this thesis does not consider the development or adoption of a SAT-based formulation and instead focuses specifically on extending the existing branch-and-bound framework to the Minimum-Size Decision Tree problem.

Witty

Staus et al. (2025) proposed *Witty*, an algorithm for constructing minimum-size perfect decision trees. Instead of searching directly over all possible decision trees, *Witty* represents the search using witness trees (Komusiewicz et al. 2023). Instead of branching on all possible splits, the algorithm starts with a single leaf and repeatedly considers all possible one-step refinements of the tree whenever it encounters an example that is classified incorrectly. Such an example is referred to as a *dirty* example. For each refinement, the algorithm introduces a new split that correctly classifies the dirty example while preserving the correctness of previously classified examples.

To further reduce the search space, Staus et al. (2025) introduce two lower bounds. The first is the Improvement Lower Bound (ImpLB), which estimates the minimum number of additional refinements required to classify all remaining dirty examples correctly. This lower bound models the problem as a Set Cover instance and computes a greedy approximation at every search node. The second is the Pair Lower Bound (PairLB), which considers conflicting pairs of examples belonging to different classes that are assigned to the same leaf. The lower bound is obtained by solving a linear programming relaxation of the corresponding Set Cover problem. Since solving this relaxation is computationally expensive, the PairLB is only computed once at the root node.

The lower bounds introduced by Staus et al. (2025) are directly relevant to this thesis. In particular, the Pair Lower Bound is reused as one of the pruning techniques within our branch-and-bound framework. In addition, we introduce a simpler Pair-size Lower Bound that can be computed efficiently at every search node.

Although *Witty* is one of the state-of-the-art algorithms for the MSDT problem, it has two important limitations. First, it does not provide anytime performance. If the search is terminated before proving optimality, no perfect decision tree is returned. Second, the algorithm searches over the space of witness trees rather than using a branch-and-bound framework, making it difficult to incorporate many of the pruning techniques that have been developed for continuous features.

2.2. CODTree

The work presented in this thesis builds upon the branch-and-bound framework for continuous features introduced in the previous sections. In particular, our approach extends CODTree (van den Bos 2025) to solve the Minimum-Size Decision Tree problem. CODTree was selected as the foundation because its modular architecture allows the optimization objective and search strategy to be adapted while retaining the underlying branch-and-bound framework. Moreover, it already incorporates techniques specifically designed for datasets with continuous features, making it well-suited to the setting considered in this thesis. Our extension therefore reuses several of these techniques, including sub-interval pruning, while adapting the framework to minimize decision-tree size.

van den Bos (2025) proposed CODTree, a branch-and-bound framework for constructing optimal decision trees with continuous features. Unlike previous methods that focus on a single search strategy or objective, such as depth-first or best-first search, CODTree is designed as a general framework that supports multiple search strategies and both classification and regression objectives. The algorithm searches for the optimal decision tree while directly considering continuous feature thresholds, avoiding the need to discretize the data beforehand.

The framework combines several techniques proposed in earlier work. It uses branch-and-bound search together with lower and upper bounds to prune the search space efficiently. In addition, it incorporates pruning methods developed for continuous features, such as threshold interval pruning, and includes a specialized solver for shallow trees to reduce the computational cost of small subproblems. Since different search strategies influence both runtime and memory usage, the framework is designed so that different traversal strategies can be evaluated within the same algorithm.

Due to its modular design and support for different search strategies, CODTree was chosen as the basis for this thesis. Rather than constructing optimal decision trees under a depth constraint, this thesis extends the framework to solve the Minimum-Size Decision Tree problem. While the underlying branch-and-bound framework is retained, the objective function, lower bounds, upper bounds, and search procedure are adapted to construct the smallest perfect decision tree. This allows pruning techniques developed for continuous features to be combined with lower bounds developed for minimum-size perfect decision trees.

2.2.1. Search Strategies

Besides the algorithm itself, the order in which the search space is explored has a large influence on the performance of a decision tree algorithm. Different search strategies make different trade-offs between memory usage, anytime performance, and the time required to prove optimality.

van den Bos (2025) compares several search strategies for branch-and-bound algorithms that are based on three operations: selecting a search node, expanding that node, and backtracking the resulting bounds. The evaluated strategies include depth-first search, global best-first search, and AND/OR best-first search.

Depth-first search explores one branch of the search tree completely before considering alternatives. This allows the algorithm to quickly obtain tight upper bounds, but often results in a longer time to optimality because other promising parts of the search space are explored much later.

Global best-first search expands the search node with the best priority according to a heuristic. This allows the algorithm to focus on the most promising subproblems, but requires considerably more memory because many partially explored nodes must remain in memory.

AND/OR best-first search exploits the AND/OR structure of the search graph. Rather than selecting nodes from a single global priority queue, it expands the subproblem that currently determines the global lower bound. This often leads to fewer unnecessary node expansions than global best-first search.

van den Bos (2025) shows that for the considered datasets and experimental setup, global best-first search with a heuristic that selects smaller subproblems first and better bounds second performs best in terms of time to the optimal solution and anytime performance.

Since this thesis extends a branch-and-bound framework, the choice of search strategy is an important part of the algorithm. In chapter 4, we compare several search strategies and introduce a memory-aware hybrid strategy that combines the advantages of AND/OR search and depth-first search.

2.3. Research Gap

Existing work on decision tree construction can broadly be divided into two directions. The first direction focuses on finding optimal decision trees under a depth constraint. Methods such as DL8.5, MurTree, STreeD, and ConTree introduce efficient pruning techniques, caching strategies, and lower bounds that significantly reduce the search space. However, these algorithms optimize objectives based on classification error and are not designed to solve the Minimum-Size Decision Tree problem.

The second direction focuses specifically on constructing minimum-size perfect decision trees. Witty introduces a search procedure and lower bounds that are tailored to this problem. However, it does not use the branch-and-bound framework employed by algorithms such as ConTree and does not provide anytime performance.

This thesis combines ideas from both research directions. We extend the CODTree branch-and-bound framework to solve the Minimum-Size Decision Tree problem for datasets with continuous features. To

improve the efficiency of the search, we incorporate lower bounds developed for perfect decision trees together with pruning techniques for continuous features. In addition, we introduce a randomized CART upper bound that provides anytime performance and a memory-aware search strategy that balances search efficiency and memory usage.

Witty and our method solve the same optimization problem, but search for the solution in fundamentally different ways. Witty operates on witness trees and expands them when misclassified examples are encountered. In contrast, our method formulates MSDT within CODTree’s branch-and-bound search framework, where subproblems correspond to subsets of the data induced by feature-threshold decisions. This allows us to combine MSDT-specific lower bounds with pruning techniques developed specifically for continuous features, while a CART-based upper bound maintains a feasible perfect tree throughout the search.

3

Preliminaries

This chapter covers the key concepts and notation that we use throughout the thesis. We start by looking at how data is represented through features, feature vectors, and datasets, and then introduce the decision tree model, which is central to our work. After that, we discuss the Minimum-Size Decision Tree (MSDT) problem, which is the main optimization task we focus on. Next, we describe CODTree, a branch-and-bound algorithm that forms the basis of our approach, and explain how it searches and prunes the space of possible trees. Finally, we introduce the set cover problem, which is important for computing lower bounds that we use later. Together, these preliminaries provide the foundation for the methods and results discussed in the following chapters.

3.1. Notation and Problem Definition

Let $f \in F$ be a feature that encodes information about an object. A feature may take *binary*, *categorical*, or *continuous* values depending on its domain. Categorical features can be transformed into binary features using *one-hot encoding*: for each category c in the domain of f , a binary feature f_c is introduced that takes value 1 if $x[f] = c$ and 0 otherwise. In this thesis, we primarily focus on continuous features.

A *feature vector* is a vector of feature values. Let x denote a feature vector and let $y \in Y$ denote its class label. We denote $x[f]$ as the value of feature f in x . Although classes can in general be continuous or discrete, in this work we consider binary classification, i.e., $Y = \{0, 1\}$.

A *dataset* $\mathcal{D}$ is a set of labeled examples (x, y) . We assume that $\mathcal{D}$ does not contain two examples with identical feature vectors but different class labels, since no decision tree can correctly classify both examples. For each feature $f \in F$, we let W^f be a smallest set of *thresholds* such that, for every pair of examples $(x_1, y_1), (x_2, y_2) \in \mathcal{D}$ with $x_1[f] < x_2[f]$, there exists a threshold $w \in W^f$ satisfying $x_1[f] \leq w < x_2[f]$. Such a smallest set is generally not unique. In practice, we construct W^f by sorting the distinct values of feature f and taking the midpoint between each pair of consecutive values.

A *decision tree* is a binary tree T that defines a classification function $T : X \rightarrow Y$, with X being the set of examples. Each internal node is labeled with a cut, and each leaf is assigned a class in Y . A cut is defined as a pair (f, w) , where $f \in F$ is a feature and $w \in W^f$ is a threshold, and the associated test is $x[f] \leq w$. A leaf is pure if $\forall (x_i, y_i), (x_j, y_j) \in D : y_i = y_j$ with D being the partitioned dataset remaining in the leaf. For a given feature vector x , the tree is evaluated by starting at the root and recursively applying the test at each internal node until a leaf is reached; the class assigned to that leaf is then $T(x)$. We let $|T|$ denote the size of the tree, defined as the number of internal (decision) nodes in T .

A decision tree T is a *perfect decision tree* for $\mathcal{D}$ if for every example $(x, y) \in \mathcal{D}$ we have $T(x) = y$. In the *Minimum-Size Decision Tree (MSDT)* problem, we are tasked with finding a perfect decision tree of minimum size (i.e., $\min |T|$). The MSDT problem can be formally defined as:

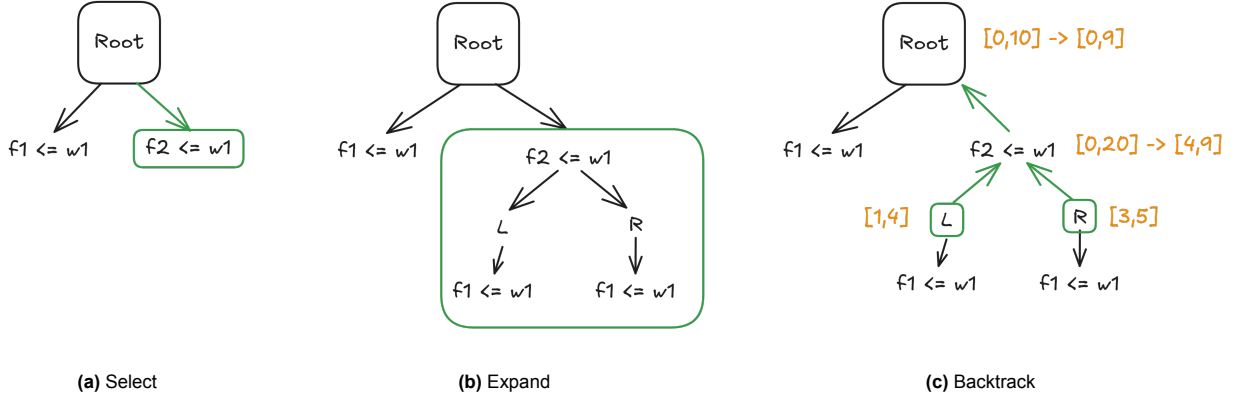


Figure 3.1: Illustration of the search graph during a single iteration of CODTree. (a) selecting a feature test based on the search strategy, (b) expanding the left and right nodes of the feature test, (c) backtracking the bounds on the loss. Repeating these three operations until the search graph is fully explored yields the optimal tree (van den Bos 2025).

$$MSDT(D) = \begin{cases} 0 & \text{if } \forall (x_i, y_i), (x_j, y_j) \in D : y_i = y_j \\ \min_{f \in F, w \in W^f} \left(1 + MSDT(D(f \leq w)) + MSDT(D(f > w)) \right) & \text{otherwise} \end{cases} \quad (3.1)$$

where F is the set of available features, W^f is the set of possible thresholds for feature f with respect to D , and $D(f \leq w)$ and $D(f > w)$ denote the subsets of D induced by splitting on feature f at threshold w . This recursion is finite because every threshold $w \in W^f$ lies between two consecutive distinct values of feature f . Consequently, both resulting subsets are nonempty and contain strictly fewer examples than D . Thus, along every branch of the recursion, the number of examples in the current subproblem strictly decreases until all remaining examples have the same class label and the base case is reached.

3.2. CODTree

In this work, we use CODTree (van den Bos 2025) as the basis for our approach. CODTree is a specialized branch-and-bound algorithm that finds optimal classification and regression trees with continuous features for an arbitrary maximum depth.

The search space is represented as a search tree where each node corresponds to a feature test that splits the data into a left and right branch. A node stores the subset of training data that reaches it, together with the remaining feature tests that can still be applied. This formulation is similar to the state-space representations used in dynamic programming methods such as DL8.5 (Aglin, Nijssen, and Schaus 2020) and MurTree (Demirović et al. 2022), where subproblems correspond to subsets of the data defined by a sequence of feature tests.

The CODTree search procedure consists of three core operations: **Select**, **Expand**, and **Backtrack**. These operations iteratively explore the search graph while maintaining lower and upper bounds on the loss to prune suboptimal branches.

The algorithm starts with a search graph containing only a root node, which maintains a priority queue of candidate feature tests. The three operations are then repeatedly executed until the optimal tree is found.

First, we **Select** a path originating from a feature test in the root priority queue. A path is a sequence of tuples (N, f_k, w) , where N is a node and the tuple represents applying the feature test $f_k \leq w$. The path, therefore, describes the sequence of splits from the root to the selected leaf node. The path in Figure 3.1a consists of a single step: $(Root, f_2, w_1)$.

Second, we **Expand** the selected leaf by applying the chosen feature test to its data, splitting the dataset into left and right subsets. These subsets initialize the corresponding child nodes, each of which main-

tains its own remaining feature tests. In Figure 3.1b, both nodes have a single remaining feature test: $f_1 \leq w_1$.

Finally, we **Backtrack** along the selected path from the leaf to the root to update the bounds on the achievable loss. During this process, feature tests whose upper bound falls below the current best lower bound can be pruned. In Figure 3.1c, the left and right nodes obtain lower bounds of one and three, and upper bounds of four and five, respectively. These bounds are combined to obtain a lower bound of four and an upper bound of nine for their parent node. At the root, the upper bound of the feature test can be updated, while the lower bound remains unchanged because another feature test may still yield a smaller loss.

By repeatedly performing these operations, CODTree incrementally explores the search graph while pruning suboptimal branches, eventually yielding the optimal decision tree. This results in the main loop given in Algorithm 1

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

```

1:  $Root \leftarrow \text{ConstructNode}(D, d)$ 
2: while  $LB(Root) < UB(Root)$  do
3:    $Path \leftarrow \text{Select}(Root)$ 
4:    $(N, f_k, w) \leftarrow \text{Head}(Path)$ 
5:    $\text{Expand}(N, f_k, w)$ 
6:   while  $Path$  is not empty do
7:      $(N, f_k, w) \leftarrow \text{Head}(Path)$ 
8:      $\text{Backtrack}(N, f_k, w)$ 
9:      $Path \leftarrow \text{Tail}(Path)$ 
10:  end while
11: end while
12: return  $UB(Root)$ 
```

3.3. Set Cover as a Lower Bound

Staus et al. (2025) use the *pair lower bound*, which relies on the *set cover* problem. In this problem, the input consists of a universe U and a family S of subsets of U . The objective is to determine the smallest integer k for which there exists a subfamily $S' \subseteq S$ of size exactly k whose union equals the universe U .

Example 1. Let $U = \{1, 2, 3, 4, 5\}$ and let $S = \{S_1, S_2, S_3\}$ where $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 4\}$, $S_3 = \{3, 4, 5\}$. Then $S' = \{S_1, S_3\}$ is a set cover of size 2, since $S_1 \cup S_3 = U$.

In the pair lower bound, we consider pairs of examples $(x_i, y_i), (x_j, y_j) \in \mathcal{D}$ that are assigned to the same leaf of the current tree but have different class labels, i.e., $y_i \neq y_j$. Let U denote the set of all such conflicting pairs.

For a split defined by a cut (f, t) , we define its *pairsplit set* $S_{(f,t)} \subseteq U$ as the set of all pairs $((x_i, y_i), (x_j, y_j)) \in U$ that are separated by the split. Equivalently, these are the pairs with different labels that are no longer assigned to the same leaf after applying the split.

Let $S = \{S_{(f,t)} \mid f \in F, w \in W^f\}$ be the family of pairsplit sets corresponding to all possible splits. Then (U, S) defines a set cover instance, where the goal is to select a minimum number of splits whose pairsplit sets cover all pairs in U . The size of an optimal solution to this instance yields a lower bound on the number of splits required to obtain a perfect decision tree.

Example 2. Consider the dataset illustrated in Figure 3.2a.

To perfectly classify these examples, we must separate every example of class 0 from the examples of class 1. The set of conflicting pairs to be covered is:

$$U = \{(x_1, x_3), (x_1, x_4), (x_2, x_3), (x_2, x_4)\}$$

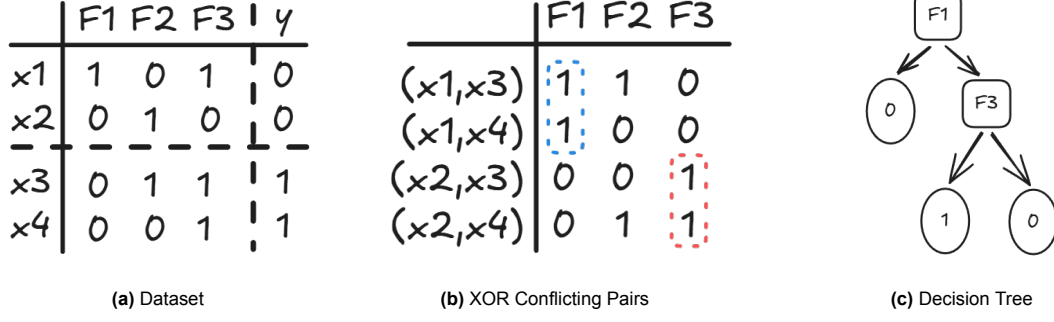


Figure 3.2: Illustration of a (a) set cover problem, (b) transformed to a decision table, and (c) a minimum size decision tree that perfectly captures the decision table. This illustration shows that selecting features 1 and 3 in (a) is a set cover of minimum size.

By applying a XOR on the feature values of the conflicting pairs, we get a table with a '1' if the feature splits example a from example b, as shown in Figure 3.2b. Using this information, we can construct a set cover problem. The goal is to find the minimum number of features whose union is U . While two combinations cover parts of U , the minimum set cover is $S_{f_1} \cup S_{f_3}$ because:

$$S_{f_1} \cup S_{f_3} = \{(x_1, x_3), (x_1, x_4), (x_2, x_3), (x_2, x_4)\} = U$$

Thus, the minimum size set cover is 2, implying the minimum decision tree requires at least 2 splits, as shown in Figure 3.2c.

Solving the set-cover instance does not generally yield the exact minimum decision-tree size. The formulation selects a collection of splits whose combined pair-split sets cover all conflicting pairs, but it does not model where those splits occur in the tree. In particular, a split selected once in the set-cover solution may need to appear in multiple branches of the decision tree. Consequently, the set-cover optimum can underestimate the number of decision nodes and therefore provides only a lower bound. This is illustrated in Example 3.

Example 3. Consider the following binary classification dataset in Table 3.1.

Table 3.1: A binary XOR dataset for which the set-cover lower bound is not tight.

Example	f_1	f_2	Class label y
x_1	0	0	0
x_2	0	1	1
x_3	1	0	1
x_4	1	1	0

The set of conflicting pairs is

$$U = \{(x_1, x_2), (x_1, x_3), (x_2, x_4), (x_3, x_4)\}.$$

The pairs separated by each feature are

$$S_{f_1} = \{(x_1, x_3), (x_2, x_4)\}, \quad S_{f_2} = \{(x_1, x_2), (x_3, x_4)\}.$$

Since

$$S_{f_1} \cup S_{f_2} = U,$$

the minimum set cover has size 2.

However, the minimum perfect decision tree has size 3. If the root splits on f_1 , then x_1 and x_2 remain together in the branch $f_1 = 0$ and must be separated using f_2 . Likewise, x_3 and x_4 remain together in the branch $f_1 = 1$ and must also be separated using f_2 . Thus, the tree uses one split on f_1 and two separate occurrences of a split on f_2 .

Therefore,

$$\text{MSDT}(D) = 3,$$

whereas the set-cover formulation gives a lower bound of 2. The set-cover formulation is not exact because it counts whether a feature is selected, but does not account for the fact that the same feature may need to occur in multiple branches of the tree.

3.3.1. Solving the Set Cover as a Linear Program

To obtain a lower bound for the set cover problem, we can formulate it as a linear program (LP).

Formally, for each split (f, t) we introduce a variable $z_{(f,t)} \in [0, 1]$, and solve:

$$\min \sum_{(f,t)} z_{(f,t)}$$

subject to:

$$\sum_{(f,t): ((x_i, x_j) \in S_{(f,t)})} z_{(f,t)} \geq 1 \quad \forall (x_i, x_j) \in U.$$

The resulting objective value may be fractional, but it represents the minimum total “coverage” required to separate all pairs of conflicting items. Since any valid solution must use an integral number of splits, we take the ceiling of the LP optimum to obtain a lower bound.

However, preliminary experiments from Staus et al. (2025) show that computing the pair lower bound at every node of the search tree using this approach does not lead to a reduction in runtime. Therefore, in their approach, the pair lower bound is computed only once at the root to provide an initial lower bound. As an alternative, the pair lower bound can be computed on a subsampled subset of the dataset in a search node, which reduces the size of the resulting set cover instance and thus the computational cost, at the expense of a potentially weaker (less informative) lower bound.

3.3.2. A Simple Approach for the Set Cover Problem

We can compute a fast lower bound on the size of an optimal set cover by considering only the sizes of the subsets in S , while ignoring any overlap between them. The idea is to select the smallest number of subsets whose combined size is at least the size of the universe.

Formally, let $S = \{f_1, f_2, \dots, f_m\}$ be a collection of subsets of a universe U . We sort the subsets in non-increasing order of their sizes:

$$|f_a| \geq |f_b| \geq \dots \geq |f_m|.$$

We then find the smallest integer k such that

$$\sum_{i=1}^k |f_i| \geq |U|.$$

This value k is a lower bound on the size of any set cover.

Example 4. Assume that we have the same dataset as in 2. Using this, we construct a difference table as shown in Figure 3.2b. Summing the number of 1's in each column gives:

$$[|f_1| = 2, |f_2| = 2, |f_3| = 2].$$

Sorting these values (which are already equal in this case), we iteratively subtract them from the size of the universe $|U| = 4$:

$$\begin{aligned} 4 - |f_1| &= 4 - 2 = 2, \\ 4 - |f_1| - |f_2| &= 4 - 2 - 2 = 0. \end{aligned}$$

Thus, the smallest k such that the cumulative size reaches $|U|$ is $k = 2$. This implies that we need at least two decision nodes to construct a perfect tree.

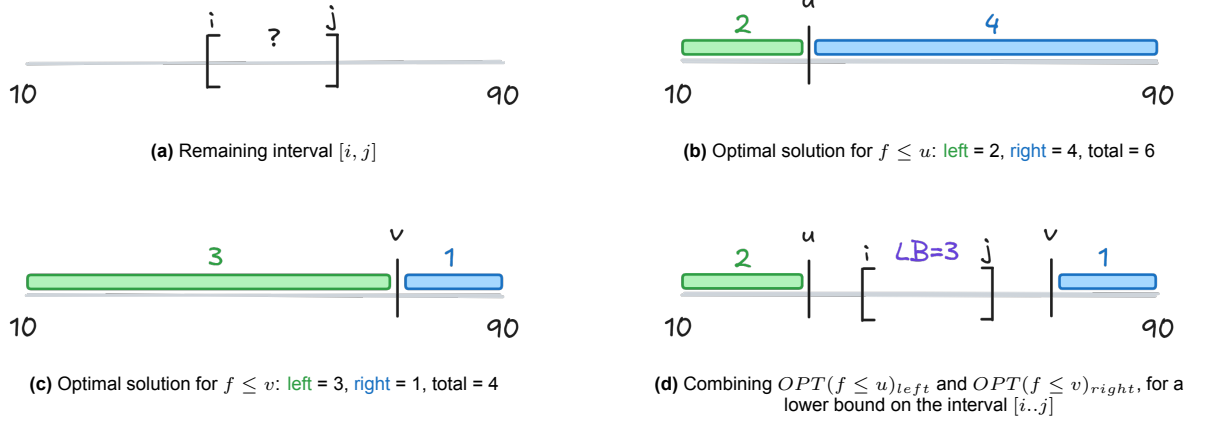


Figure 3.3: Example of sub-interval pruning providing a lower bound for the interval $[i..j]$ using the computed optimal solution for $f \leq u$ and $f \leq v$ with $u < i < j < v$

This lower bound is computationally inexpensive compared to solving the linear programming relaxation discussed in subsection 3.3.1, and in this case, gives the same lower bound.

In our approach, we refer to this lower bound as the *pair-size lower bound*.

3.4. Sub-interval Pruning

Brita, van der Linden, and Demirović (2025) introduce *sub-interval pruning*, which exploits previously computed thresholds to derive lower bounds for unexplored splits.

Consider a feature f for which the remaining candidate thresholds lie in an interval $[i, j]$, as illustrated in Figure 3.3a. Assume that we have already computed the smallest perfect decision trees for two thresholds u and v with $u < i < j < v$, as shown in Figure 3.3b and Figure 3.3c.

For any threshold $w \in [i, j]$, the split created by w produces a left and right partition that are bounded by those created by u and v . In particular, the left subtree of w contains at least the instances of the left subtree created by u , and the right subtree of w contains at least the instances of the right subtree created by v .

Therefore, a valid lower bound on the size of any decision tree using threshold $w \in [i, j]$ is given by:

$$LB(w) \geq OPT(f \leq u)_{left} + OPT(f \leq v)_{right}$$

as shown in Figure 3.3d.

4

Methodology

This chapter describes the methodology used to adapt the CODTree algorithm for solving the Minimum Size Decision Tree (MSDT) problem. We first explain how the original CODTree formulation, which optimizes misclassification under a depth constraint, is modified to instead search for the smallest decision tree that perfectly classifies the training data. Next, we describe the branch-and-bound framework used to efficiently explore the search space, including the lower and upper bound techniques that enable pruning. In particular, we present the class-count, one-off, pair, and pair-size lower bounds, together with a modified CART heuristic that provides an upper bound and enables anytime performance. Finally, we describe additional techniques used to improve efficiency, including a memory-aware search strategy, and data-reduction methods.

4.1. Finding Perfect Trees

CODTree’s primary objective is to find an *optimal decision tree*, i.e., a tree T that minimizes the *misclassification score* subject to a given maximum depth constraint. In other words, CODTree searches for the decision tree of bounded depth that achieves the lowest possible classification error on the training data.

This differs from the *Minimum Size Decision Tree (MSDT)* problem, where the goal is to find the *smallest decision tree that perfectly classifies the training data*, i.e., a tree with a misclassification score equal to zero. Unlike CODTree, MSDT does not impose a depth constraint; instead, the search space includes all trees that achieve perfect classification, and the algorithm must select the tree of minimum size.

In standard CODTree, the search is guided by the misclassification score. Lower and upper bounds for partial solutions are defined in terms of misclassification, and backtracking explores alternative branches to reduce classification errors.

To adapt CODTree for MSDT, the search criterion is modified to prioritize tree size over misclassification. Specifically, we define lower and upper bounds based on tree size, and only allow *pure leaf nodes*.

4.2. Bounds

To prune parts of the search space efficiently, we employ four *lower bound* techniques and one *upper bound* technique within a branch-and-bound framework. These bounds help determine whether a branch can potentially yield a better solution than the current best and whether it can be safely pruned.

4.2.1. Class-count Lower Bound

Any perfect decision tree for $\mathcal{D}$ must assign at least one leaf to each class in Y . Hence, it must contain at least $|Y|$ leaves. Since a binary decision tree with L leaves has exactly $L - 1$ internal nodes, it follows that any such tree has size at least $|Y| - 1$. Therefore, $|Y| - 1$ is a lower bound on the size of an optimal solution to MSDT.

	F1	F2	F3	y
x1	1	0	1	0
x2	0	1	0	0
x3	0	1	1	1
x4	0	0	1	1

(a) Dataset

	F1	F2	F3
(x1,x3)	1	1	0
(x1,x4)	1	0	0
(x2,x3)	0	0	1
(x2,x4)	0	1	1

(b) XOR Conflicting Pairs

Figure 4.1: Illustration of a (a) dataset, (b) transformed to a difference table.

$$CcLB(D) = |Y_D| - 1 \quad (4.1)$$

where $Y_D := \{y \in Y \mid \exists x \text{ such that } (x, y) \in D\}$.

4.2.2. One-off Lower Bound

The one-off lower bound uses conflicting pairs that differ only in one feature in the difference table, as shown in Figure 4.1. These features are called *one-off features*, and they are guaranteed to appear in every perfect decision tree. Due to this property, counting the number of unique one-off features gives us a lower bound for the size of the minimum-size perfect decision tree. Secondly, because these features must appear in every perfect decision tree, we prioritize selecting them in our search strategies.

Example 5. Consider a dataset and feature set as illustrated in Figure 4.1a:

$$\mathcal{D} = \{(x_1, 0), (x_2, 0), (x_3, 1), (x_4, 1)\}, F = \{[1, 0, 0, 0], [0, 1, 1, 0], [1, 0, 1, 1]\}$$

Constructing a difference table from this dataset yields the table in Figure 4.1b. From this we can derive that F1 and F3 are one-off features since the conflicting pair (x_1, x_3) is only separated by F1, and (x_2, x_3) is only separated by F3.

4.2.3. Pair and Pair-size Lower Bound

For the third and fourth lower bounds, we employ the Pair lower bound and the Pair-size lower bound as described in subsection 3.3.1 and section 3.3.2, respectively. Since the Pair lower bound is computationally expensive, and Staus et al. (2025) note that it is not beneficial to evaluate it at every search node, we compute it only at the root node. In contrast, the Pair-size lower bound is significantly less costly to evaluate, especially since we already computed the difference table at the root node. Therefore, we compute it on every search node.

We also considered the Similarity Lower Bound proposed by Demirović et al. (2022). Since the bound is theoretically weak in our setting and introduces additional computational overhead, it was not included in the final implementation. A discussion of this bound and its limitations is provided in Appendix C.

4.2.4. CART Upper Bound and Anytime Performance

Breiman et al. (1984) introduced the Classification and Regression Trees (CART) algorithm, a widely used heuristic for constructing decision trees. CART is computationally efficient, with a typical time complexity of $O(|F||D| \log |D|)$, where $|F|$ is the number of features and $|D|$ is the number of instances. However, because CART is a greedy heuristic, it does not guarantee that the resulting tree is an optimal solution to Equation 3.1.

Despite this limitation, CART can be used within our branch-and-bound framework to obtain an upper bound. Specifically, we run CART without pruning or other stopping constraints, requiring it to continue splitting until every leaf is pure. Assuming that the dataset is perfectly separable using the available

Algorithm 2 Randomized CART Upper Bound**Require:** Dataset D , feature set F , patience p **Ensure:** Best feasible perfect decision tree T^*

```

1:  $T^* \leftarrow \text{CART}(D, F)$ 
2:  $bestSize \leftarrow |T^*|$ 
3:  $noImprovement \leftarrow 0$ 
4: while  $noImprovement < p$  do
5:    $F' \leftarrow \text{RandomSubset}(F)$ 
6:    $T \leftarrow \text{CART}(D, F')$ 
7:   if  $|T| < bestSize$  then
8:      $T^* \leftarrow T$ 
9:      $bestSize \leftarrow |T|$ 
10:     $noImprovement \leftarrow 0$ 
11:   else
12:      $noImprovement \leftarrow noImprovement + 1$ 
13:   end if
14: end while
15: return  $T^*$ 

```

splits, the resulting tree perfectly classifies the dataset and therefore provides a valid upper bound on the size of an optimal tree.

An additional advantage of this approach is that it naturally provides *anytime performance*. If the modified CART procedure is executed, it produces a perfect decision tree for the dataset. This tree can be returned as a valid solution if the search is terminated early. In contrast, some state-of-the-art algorithms for the minimum-size perfect decision tree problem, such as Witty (Staus et al. 2025), do not inherently provide such intermediate feasible solutions.

To further improve the quality of the upper bound, we employ a variant of CART. Rather than constructing a single tree, we repeatedly run the algorithm on different randomly selected subsets of features. This diversification allows the search to explore different tree structures and often leads to substantially smaller feasible trees. We adopt a *patience* parameter p to control the number of repetitions. After each run, the size of the best tree found so far is recorded. If no improvement is observed for p consecutive runs, the procedure terminates.

Empirically, we found that patience values between 5 and 10 provide a favorable trade-off between runtime and solution quality across the datasets considered in our experiments. Because this procedure produces tight upper bounds, we execute it only at the root node of the branch-and-bound search. The pseudocode for the Randomized CART upper bound can be found in Algorithm 2.

4.3. Memory Aware Search Strategy

The experimental evaluation presented in Appendix B shows that both the AND-OR and DFS-Prio search strategies perform well in terms of solving efficiency. However, while the AND-OR strategy generally explores the search space more effectively, it also maintains a significantly larger number of open nodes, resulting in substantially higher memory consumption. On several datasets, this led to memory exhaustion before the search could be completed.

To address this limitation, we combine the strengths of both strategies in a memory-aware hybrid search approach. The solver primarily employs the AND-OR strategy while memory usage remains below a predefined threshold of 75% of the available memory. Once this threshold is exceeded, the search temporarily switches to DFS-Prio. Since DFS-Prio explores the search space in a depth-first manner, it requires considerably less memory and helps reduce the number of active nodes stored in memory.

By dynamically alternating between these two strategies, the solver benefits from the strong search guidance of AND-OR during most of the search while retaining the low-memory characteristics of DFS-Prio when memory pressure becomes high. This hybrid approach allows the search to remain efficient without risking memory exhaustion on larger instances.

4.4. Data Reduction

As a preprocessing step, we employ the same data-reduction strategies as in Staus et al. (2025). This includes removing duplicate examples with identical feature values, removing dimensions where all examples share the same value, removing cuts that induce identical partitions of the dataset, removing redundant extreme thresholds within dimensions, and merging pairs of dimensions that share a consistent ordering of examples.

5

Experiments

This chapter presents an empirical evaluation of the proposed modifications to the CODTree algorithm. Our experiments are designed to investigate three main aspects:

1. **Comparing search strategies:** We run different search strategies against each other to evaluate which search strategy is the best for finding minimum-size perfect trees.
2. **Comparison with a of the state-of-the-art:** We benchmark our approach against Witty (Staus et al. 2025), one of the state-of-the-art algorithms for constructing minimum-size perfect decision trees.
3. **Effectiveness of the bounds:** We evaluate how the lower and upper bound methods influence the efficiency of the search process.

5.1. Experiment Setup

All experiments are conducted on a machine with a Ryzen 5600X CPU (6 cores, 12 threads, 3.7 GHz) and 32GB of RAM. We use datasets that we obtained from OpenML. Here, we searched for datasets that contain only continuous features, as CODTree is optimized for them. To keep the runtime of the experiments within reason, we then sampled these datasets to include only the first 100 examples, a similar approach to Staus et al. (2025). The characteristics of each dataset are described in Appendix A. The following sections present the detailed experimental setup, results, and analysis for each aspect of our study.

5.2. Comparing Search Strategies

In this section, we compare different search strategies by collecting intermediate lower and upper bounds on the sampled `diggle_table_a1`, `german-credit-data-credibility`, and `pc1` datasets under a time limit of 20 minutes and a memory limit of 4 GB.

In Figure 5.1, we present results for four search strategies: AND-OR, DFS-Prio, AND-OR-DFS-Prio (memory aware) from section 4.3, and BFS-balance-small-LB. We have chosen these search strategies since AND-OR and DFS-Prio outperformed the other search strategies by a large margin, and the BFS-balance-small-LB is the search strategy that performed best according to van den Bos (2025). The results for all evaluated strategies are provided in Appendix B. Below, we will give a brief recap of how the strategies work.

AND-OR search works by first prioritizing nodes with the lowest lower bound. As a secondary criterion, it prefers nodes that have already been expanded over those that have not. When expanding a node, it selects the child with the highest upper bound.

DFS-Prio similarly uses a two-level priority scheme, but with reversed expansion bias. It first prioritizes expanded nodes over unexpanded nodes, and then breaks ties using the lowest lower bound. When selecting a child, it again chooses the one with the highest upper bound.

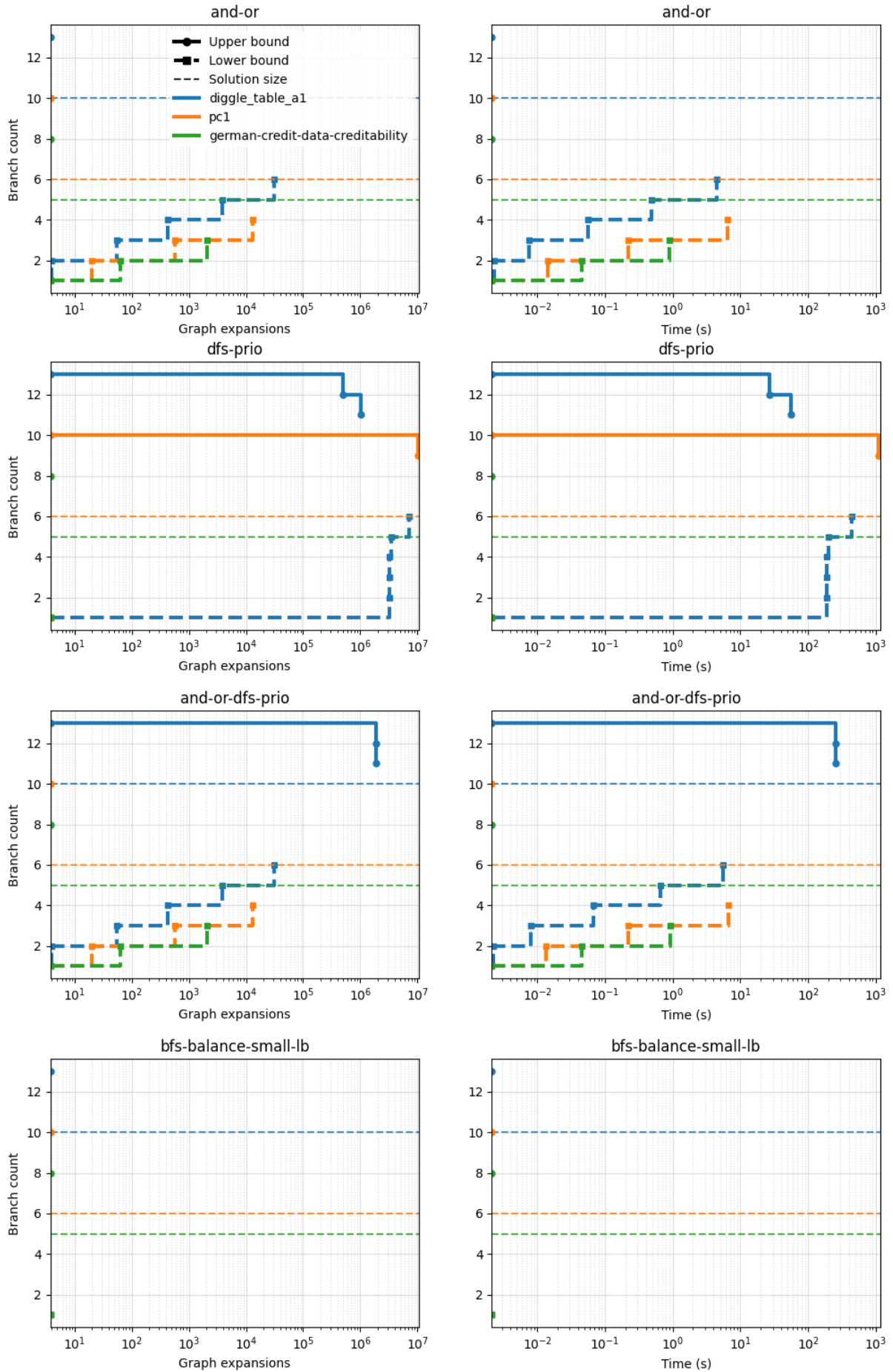


Figure 5.1: The AND-OR, DFS-Prio, AND-OR-DFS-Prio (memory aware), and BFS-balance-small-LB search strategies on the sampled datasets diggle_table_a1, german-credit-data-credibility, and pc1 with a time limit of 20 minutes and a memory limit of 4 GB.

BFS-balance-small-LB is a best-first search strategy that ranks nodes using a combined scoring function. Each node is assigned a score equal to its lower bound plus how many instances reach that node. The search always expands the node with the smallest score. This biases the search toward nodes that are both low-cost in terms of the current lower bound and small in terms of support, thereby favoring more compact subproblems.

From Figure 5.1, we observe that both the AND-OR and DFS-Prio search strategies significantly outperform the other strategies. This is mainly because both approaches strongly prioritize low lower bounds when deciding which part of the search space to explore next, enabling the algorithm to prune large parts of the search space. DFS-Prio applies this prioritization less aggressively than AND-OR, as it first completes the exploration of a subtree before moving on to another node with a lower bound. As a result, DFS-Prio is less prone to running into memory issues compared to AND-OR. We also observe that the AND-OR-DFS-Prio strategy as described in section 4.3, combines advantages of both strategies. It is able to improve the global lower bound quickly through AND-OR search, but does not run into memory issues anymore due to the switch to DFS-Prio at 75% of memory usage.

5.3. Comparison with a State-of-the-Art Method

In this section, we compare our modified CODTree algorithm with Witty, a state-of-the-art method for constructing minimum-size perfect decision trees introduced by Staus et al. (2025).

We assess the performance of both algorithms using three metrics. First, we measure runtime to evaluate overall computational efficiency. Second, we record the number of expansions in the search graph, which reflects search efficiency: fewer expansions indicate that the algorithm more effectively prioritizes search nodes and prunes the search space using the available information. Third, we measure the anytime performance of both algorithms.

We run both algorithms with a timeout of 30 minutes and a memory limit of 4 GB. For CODTree, we evaluate the AND-OR, DFS-Prio, and BFS-balance-small-LB search strategies. For Witty, we enable all features by using algorithm ID 5, allocate 30 seconds to the upper-bound procedure, and set the maximum tree size to -1 , as this parameter is unused in the selected configuration.

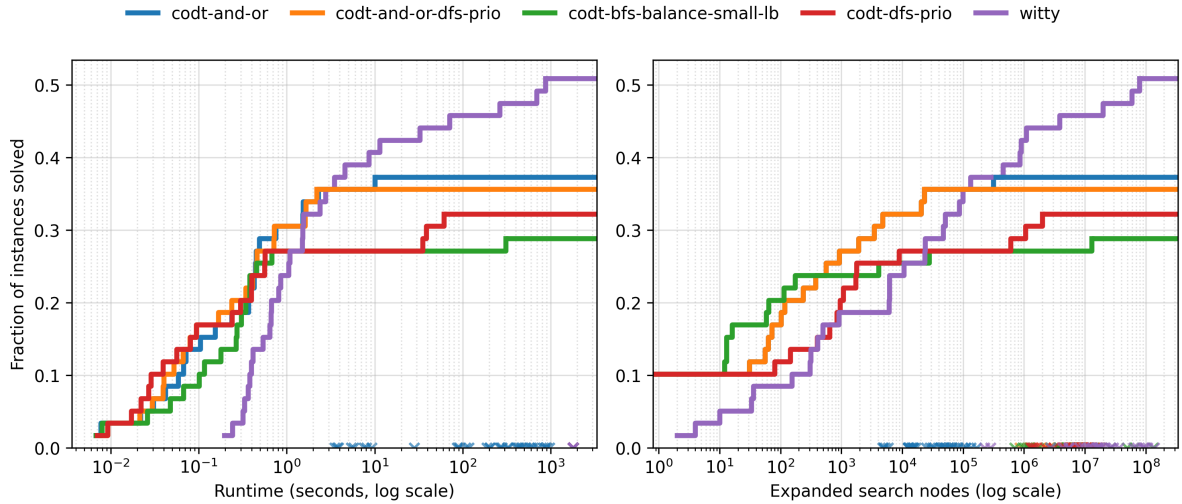


Figure 5.2: Runtime in seconds and number of search-graph expansions versus the fraction of instances solved by CODTree using the AND-OR, AND-OR-DFS-Prio (memory-aware), BFS-balance-small-LB, and DFS-Prio strategies, and by Witty.

Figure 5.2 presents the aggregate runtime and search-expansion results. Regardless of the search strategy, CODTree performs well on the easier instances and initially solves a larger fraction of the benchmark set than Witty. As the instances become harder, however, Witty scales better and ultimately solves more instances within the time and memory limits.

Table 5.1 provides a more detailed comparison on selected instances for which Witty proves optimality. On many of the smaller instances, CODTree is faster than Witty, sometimes by more than an

Dataset	Witty		CODT		Dataset	Witty		CODT	
	T	Time	T	Time		T	Time	T	Time
analcadata_bankruptcy	4	0.41	4	0.04	iris	1	0.19	1	0.01
analcadata_election2000	2	0.32	2	0.03	kc1-top5	3	4.59	3	2.19
autoprice	3	0.54	3	0.05	kc2	2	0.33	2	0.09
breast-w	5	0.36	5	1.62	kc3	6	8.60	9	1800
chatfield_4	6	693	7	1800	oil_spill	3	1.05	3	0.46
chscase_adopt	8	0.38	8	0.72	parkinsons	3	0.67	3	0.41
chscase_vine1	5	1.08	5	0.45	pc1	6	264	9	1800
chscase_vine2	24	71	25	1800	piechart2	3	0.64	3	0.08
CMSC ¹	4	11	4	1.64	pyrim	7	1.52	9	1800
datatrieve	3	3.49	3	0.02	rabe_266	9	0.24	9	0.01
diggie_table_a1	10	872	11	1800	sonar	2	0.66	2	0.17
divorce_prediction	2	2.38	2	0.04	strikes	1	0.39	1	0.07
GCDC ²	5	2.75	8	1800	vehicle	4	1.50	4	0.71
GCDC-2 ³	5	1.55	6	1800	wind_correlations	3	0.81	3	0.24
hutsof99_child_witness	6	33	6	1800	wine	3	0.86	3	0.34

Table 5.1: Comparison of Witty and CODTree using the AND-OR-DFS-Prio strategy. Tree size $|T|$ is measured by the number of branch nodes, and runtime is reported in seconds. A runtime of 1800 seconds indicates that the algorithm reached the time limit. Bold indicates the lower runtime for an instance.

order of magnitude. On the more difficult instances, CODTree frequently reaches the time limit before proving optimality, whereas Witty completes the search. Nevertheless, the feasible trees returned by CODTree are generally close in size to the optimal trees found by Witty. For example, on `chatfield_4`, `chscase_vine2`, and `diggie_table_a1`, the tree returned by CODTree is only one branch node larger than the optimum. On `pc1`, `pyrim`, and `GCDC`, the difference is three branch nodes. Thus, although CODTree is less effective at proving optimality on difficult instances, it often still returns a high-quality feasible solution within the available time.

Dataset	Witty	CART UB	CODT	Dataset	Witty	CART UB	CODT
appendicitis_test_edsa	–	15	13	ilpd-numeric	–	20	17
arcene_seed_0	–	15	13	ionosphere	–	9	9
arcene_seed_3	–	18	13	jedit_4_2_4_3	–	25	25
chscase_census3	–	25	19	kc1-binary	–	17	17
chscase_census5	–	26	19	megawatt1	–	11	11
chscase_census6	–	25	19	mindcave2	–	22	22
costamadre1	–	16	16	no2	–	23	20
credit_approval	–	22	22	pizzacutter1	–	13	13
dataset_credit-approval	–	16	16	rabe_265	–	10	10
dataset_credit-g	–	25	23	sleuth_ex2015	–	14	11
diabetes	–	18	14	tuningsvms	–	15	15
disclosure_x_noise	–	33	26	environmental	–	30	26
disclosure_z	–	34	28	hamster	–	14	13
glass	–	16	11	wisconsin	–	23	22
heart-statlog-uci	–	24	15				

Table 5.2: Tree sizes on instances for which both Witty and CODTree reached the time limit. Witty did not return a feasible tree, whereas CODTree returned a perfect decision tree for every instance. The CART upper bound is included to show the improvement obtained by CODTree during the subsequent branch-and-bound search. Tree size is measured by the number of branch nodes, and a dash indicates that no feasible solution was returned.

The results in Table 5.2 demonstrate an important advantage of CODTree’s anytime behavior. On every instance in this table, both algorithms reach the time limit. However, Witty does not return a feasible tree, whereas CODTree returns a perfect decision tree for every instance. Furthermore, CODTree often improves substantially upon its initial CART upper bound. For example, it reduces the tree size from 33 to 26 branch nodes on `disclosure_x_noise`, from 24 to 15 on `heart-statlog-uci`, and from 26 to 19 on `chscase_census5`. These improvements show that CODTree does more than merely retain the

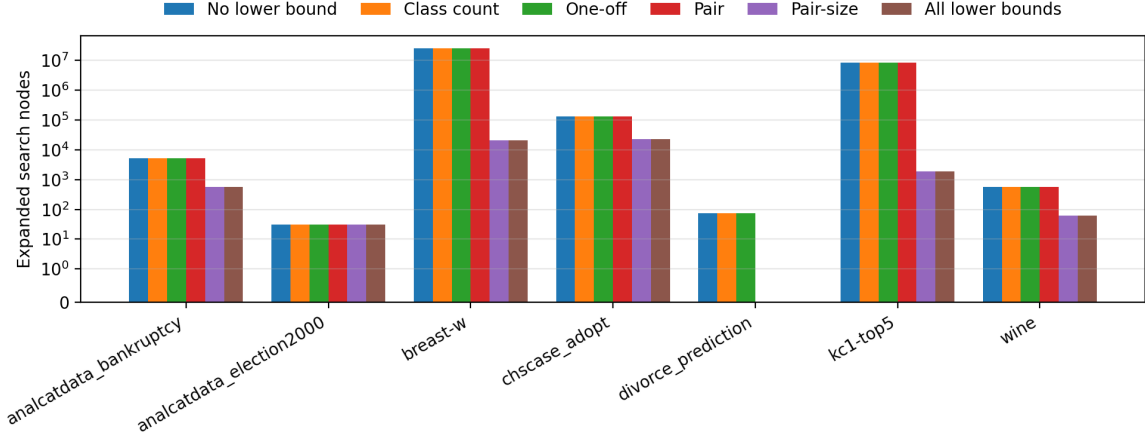


Figure 5.3: Expanded search nodes for each lower bound in isolation, compared with no lower bounds and all lower bounds.

solution produced during CART initialization: its branch-and-bound search continues to find smaller feasible trees throughout the available computation time. Consequently, even when CODTree cannot prove optimality, it still performs useful optimization and returns a substantially improved solution on 17 out of 29 datasets. In contrast, Witty returns no feasible solution for these instances.

There is also a trade-off between the CODTree search strategies. The AND-OR strategy generally solves easier instances faster, but it runs out of memory on harder instances. The DFS-Prio and BFS-balance-small-LB strategies use memory more conservatively, but consequently reach the time limit more often.

Although CODTree solves fewer difficult instances to optimality than Witty, it requires substantially fewer search-graph expansions. In particular, CODTree solves approximately 10% of the benchmark instances without performing any search-graph expansions. This occurs when the pair lower bound and the CART upper bound, described in subsection 3.3.1 and section 4.2.4, respectively, coincide. In that case, optimality is established immediately, and no further search is required.

5.4. Effectiveness of the Bounds

In this section, we investigate the effectiveness of the lower and upper bound methods described in section 4.2. To this end, we compare the number of expanded search nodes for different bound configurations.

When evaluating a lower bound, only that specific lower bound is enabled, while all other lower bounds are disabled. We compare each lower bound against a configuration without any lower bounds and against a configuration in which all lower bounds are enabled. This allows us to measure both the individual contribution of each lower bound and their combined effect on search efficiency.

The upper bound is evaluated separately by comparing the algorithm with and without the CART-based upper bound.

Lower Bounds

From Figure 5.3, we observe that the pair-size lower bound is by far the most effective lower bound. Across almost all datasets, enabling this bound reduces the number of expanded search nodes by one to several orders of magnitude compared to using no lower bound. This strong effect is likely due to the bound being computationally inexpensive and therefore applicable at every search node while still producing tight lower bounds, enabling the algorithm to effectively prune parts of the search space.

The effect is particularly visible for the `kc1-top5` dataset, where the number of expanded nodes decreases from approximately 10^7 to around 10^3 , and for `analcatdata_bankruptcy` and `wine`, where reductions of roughly an order of magnitude are observed. For `breast-w`, the pair-size lower bound also substantially reduces the search effort, although the reduction is less dramatic. Furthermore, for

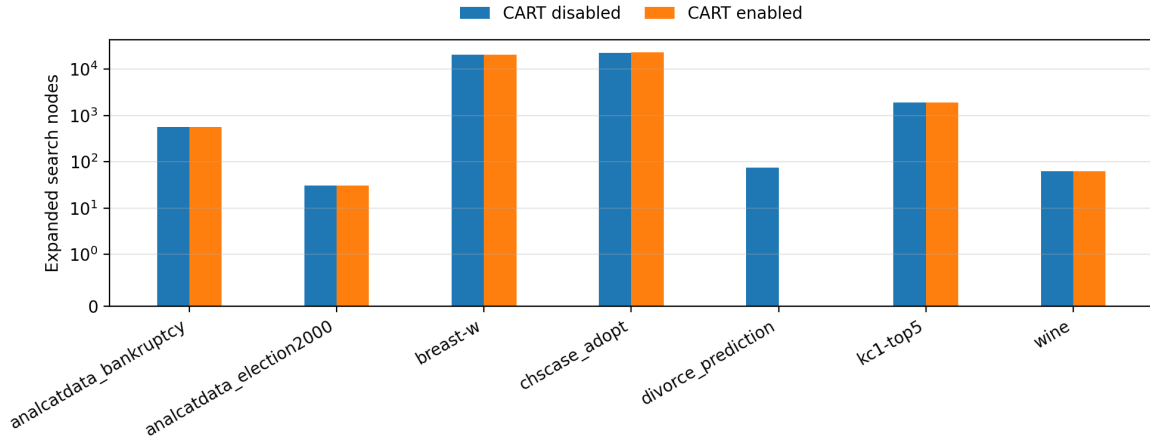


Figure 5.4: Search graph expansions with and without the CART upper bound from section 4.2.4

the `divorce_prediction` dataset, activating either the pair lower bound or the pair-size lower bound eliminates the need for any search expansions, indicating that the optimal solution can be certified directly at the root node.

Although the pair lower bound provides a stronger lower bound at the root node, it does not lead to additional pruning in our implementation. Since the bound is only computed for the root node and not for descendant nodes, it cannot be used to eliminate subproblems during the branch-and-bound search. Its effect is therefore limited to improving the global lower bound and reducing the reported optimality gap.

The class-count lower bound described in subsection 4.2.1 and the one-off lower bound described in section 4.2.2 have virtually no impact on the considered datasets. The class-count lower bound provides only weak information because all datasets involve binary classification, resulting in a lower bound of one and therefore offering little additional information. Similarly, the one-off lower bound does not enable any pruning. This is expected for all datasets except `chscase_adopt`, as they contain no one-off features. Even for `chscase_adopt`, which contains two one-off features, the bound does not lead to measurable pruning.

CART Upper Bound

From Figure 5.4, we observe that the CART-derived upper bound has a negligible effect on the search effort. The number of expanded nodes remains almost unchanged across all datasets when the upper bound is enabled. The only visible difference is for `divorce_prediction`, where the search is eliminated when the CART upper bound is available, while a small number of nodes are still expanded without it. Overall, these results suggest that the CART upper bound is generally not tight enough to significantly improve pruning on the benchmark instances considered. It does, however, provide good anytime performance.

6

Conclusion

This thesis presents an extension of the CODTree framework to solve the Minimum-Size Decision Tree (MSDT) problem, focusing on the construction of perfect decision trees with continuous features. By reformulating the objective from minimizing misclassification to minimizing tree size under perfect classification, the approach adapts branch-and-bound search to a fundamentally different optimization criterion.

To improve the efficiency of the search, several complementary lower bounds were integrated, namely the class-count (subsection 4.2.1), one-off (section 4.2.2), pair (subsection 3.3.1), and pair-size (section 3.3.2) lower bounds. A randomized CART procedure was additionally introduced to obtain strong feasible upper bounds (section 4.2.4). Besides improving pruning, this upper-bound procedure ensures that the algorithm maintains a valid perfect decision tree throughout the search, thereby improving the anytime performance. Finally, a memory-aware hybrid search strategy was proposed that primarily uses AND/OR search but switches to DFS-Prio when memory consumption becomes too high.

The experimental results show that the adapted CODTree algorithm performs well on smaller and easier instances. On these instances, it can find and prove optimal solutions quickly and often requires relatively few search-graph expansions. The experiments also indicate that the pair-size lower bound provides the best practical trade-off between computational cost and pruning effectiveness among the evaluated bounds. Furthermore, the memory-aware search strategy preserves much of the strong search guidance of AND/OR search while reducing the risk of memory exhaustion.

However, the comparison with Witty shows that the proposed approach does not yet scale as well to the more difficult benchmark instances. Witty ultimately solves a larger fraction of the evaluated datasets within the time and memory limits. Nevertheless, the adapted CODTree approach has an important practical advantage: when the search is interrupted, it can return the best perfect decision tree found up to that point. In contrast, Witty may terminate without producing a valid perfect decision tree.

Overall, this work demonstrates that a branch-and-bound framework developed for optimal decision trees with continuous features can be successfully adapted to the MSDT problem. The combination of problem-specific bounds, adaptive search strategies, and strong feasible upper bounds yields a solver that is particularly useful when obtaining a valid solution quickly is important, even when optimality cannot be proven within the available resources. At the same time, the experiments indicate that stronger pruning and more specialized procedures are required to make the approach competitive on harder instances.

Future work could first focus on developing stronger yet computationally inexpensive lower bounds. The experiments in this thesis show that the pair-size lower bound offers an effective trade-off between pruning strength and computational cost and yields the strongest pruning among the evaluated lower bounds. A possible direction to improve this bound would be to retain more information about the overlap between pair-split sets without solving the complete LP relaxation. Since the pair-size lower bound only considers the number of conflicting pairs separated by each split, it may overestimate the

combined coverage of multiple splits when their pair-split sets overlap substantially. Accounting for part of this overlap could therefore strengthen the bound while remaining cheaper than solving the full LP relaxation.

A second direction is the development of a specialized solver for small or shallow subproblems. Previous work, such as `MurTree` (Demirović et al. 2022), has shown that specialized procedures for shallow subproblems can considerably reduce search time. Developing an analogous procedure for the minimum-size objective could substantially improve the efficiency of the proposed approach. One concrete approach would be to adapt the depth-two optimal-tree subroutine introduced by (Brita, van der Linden, and Demirović 2025) to the MSDT objective. Instead of minimizing classification loss under a depth constraint, the procedure would need to determine the smallest perfectly classifying tree among the shallow candidate structures. Such a routine could be invoked for subproblems restricted to depth two, avoiding the overhead of the general branch-and-bound procedure for these cases.

References

- 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* 34.04, pp. 3146–3153. doi: 10.1609/aaai.v34i04.5711.
- Amann, Julia, Alessandro Blasimme, Effy Vayena, Dietmar Frey, Vince I. Madai, and the Precise4Q consortium (2020). “Explainability for Artificial Intelligence in Healthcare: A Multidisciplinary Perspective”. In: *BMC Medical Informatics and Decision Making* 20.1, p. 310. doi: 10.1186/s12911-020-01332-6.
- Avellaneda, Florent (2020). “Efficient Inference of Optimal Decision Trees”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04, pp. 3195–3202. doi: 10.1609/aaai.v34i04.5717.
- Breiman, Leo, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone (1984). *Classification and Regression Trees*. 1st ed. Chapman and Hall/CRC. doi: 10.1201/9781315139470.
- Brita, 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. doi: 10.1609/aaai.v39i11.33210.
- 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.
- Garrett, Brandon L. and Cynthia Rudin (2023). “Interpretable algorithmic forensics”. In: *Proceedings of the National Academy of Sciences of the United States of America* 120 (41). issn: 10916490. doi: 10.1073/pnas.2301842120.
- Hyafil, Laurent and Ronald L. Rivest (1976). “Constructing optimal binary decision trees is NP-complete”. In: *Information Processing Letters* 5.1, pp. 15–17. doi: 10.1016/0020-0190(76)90095-8.
- 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*. Ed. by Luca Pulina and Martina Seidl. Cham: Springer International Publishing, pp. 501–518. isbn: 978-3-030-51825-7.
- Komusiewicz, Christian, Pascal Kunz, Frank Sommer, and Manuel Sorge (2023). “On Computing Optimal Tree Ensembles”. In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, pp. 17364–17374.
- 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*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 6150–6160.
- 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*. Ed. by Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato. Vol. 162. Proceedings of Machine Learning Research. PMLR, pp. 15255–15277.
- Narodytska, Nina, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva (2018). “Learning Optimal Decision Trees with SAT”. In: *Proceedings - 27th International Joint Conference on Artificial Intelligence, IJCAI 2018*. Association for the Advancement of Artificial Intelligence (AAAI), pp. 1362–1368. doi: 10.24963/ijcai.2018/189.
- 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*. New York, NY, USA: Association for Computing Machinery, pp. 530–539. isbn: 9781595936097. doi: 10.1145/1281192.1281250.

- 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. doi: 10.1609/aaai.v35i7.16800.
- Salzberg, Steven L. (1994). "C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993". In: *Machine Learning* 16.3, pp. 235–240. doi: 10.1007/BF00993309.
- 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. issn: 1572-9354. doi: 10.1007/s10601-023-09348-1.
- 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. doi: 10.1609/aaai.v39i19.34268.
- Van den Bos, Mim (2025). "Search Strategies for Optimal Decision Tree Classification and Regression with Continuous Features". Master's thesis. Delft University of Technology.
- Van der Linden, Jacobus G. M., Mathijs de Weerd, and Emir Demirović (2023). "Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., pp. 9173–9212.

A

Datasets

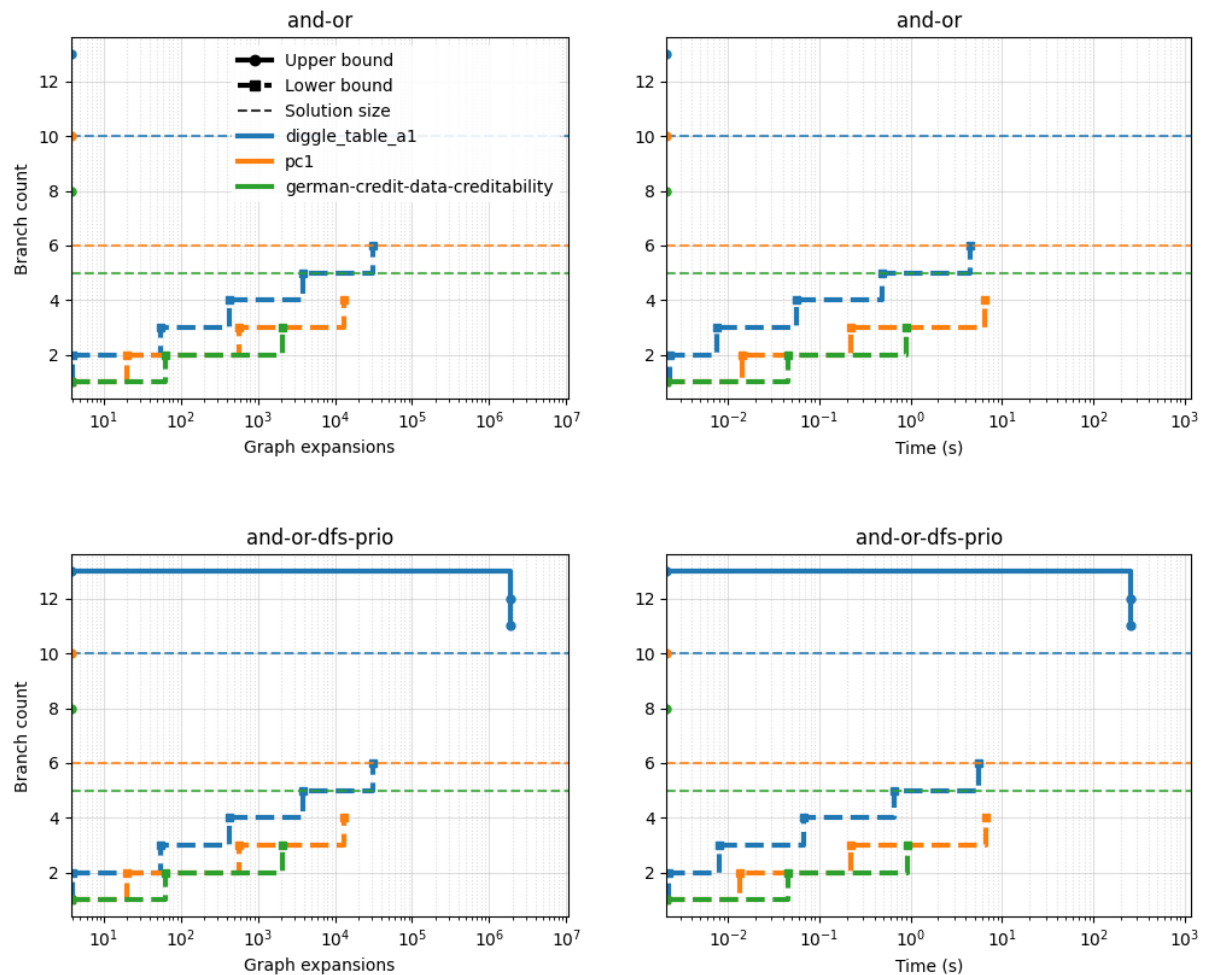
Dataset	$ D $	$ F $	$ W $	$ T $	Dataset	$ D $	$ F $	$ W $	$ T $
analcata_data_bankruptcy	50	5	237	4	ilpd-numeric	100	10	399	-
analcata_data_election	67	14	728	2	ionosphere	100	34	2039	-
appendicitis_test_edsa	100	7	501	-	iris	100	4	90	1
arcene_seed_0	100	100	4774	-	jedit_4_2_4_3	100	8	251	-
arcene_seed_3	100	100	4025	-	kc1-binary	100	94	4620	-
autoprice	100	15	443	3	kc1-top5	100	94	4620	3
breast-w	100	9	75	5	kc2	100	21	780	2
chatfield_4	100	12	1086	6	kc3	100	39	1131	6
chscase_adopt	39	2	67	8	megawatt1	100	37	1377	-
chscase_census3	100	7	678	-	mindcave2	100	39	1744	-
chscase_census5	100	7	674	-	no2	100	7	476	-
chscase_census6	100	6	593	-	oil_spill	100	49	2626	3
chscase_vine1	52	9	185	5	parkinsons	100	22	2026	3
chscase_vine2	100	2	52	24	pc1	100	21	1045	6
CMSC ¹	100	18	1782	4	piechart2	100	36	1176	3
costamadre1	100	37	1533	-	pizzacutter1	100	37	1450	-
credit_approval	100	50	214	-	pyrim	74	27	80	7
dataset_credit-approval	100	15	354	-	rabe_265	51	6	235	-
dataset_credit-g	100	20	204	-	rabe_266	100	2	19	6
datatrieve	100	8	531	3	sleuth_ex2015	60	7	369	-
diabetes	100	8	398	-	sonar	100	60	5585	2
diggie_table_a1	48	4	110	10	strikes	100	6	269	1
disclosure_x_noise	100	3	297	-	tuningsvms	100	80	5314	-
disclosure_z	100	3	297	-	vehicle	100	18	684	4
divorce_prediction	100	54	216	2	environmental	100	3	186	-
GCDC-2 ²	100	20	198	5	hamster	73	5	356	-
GCDC ³	100	71	215	5	wind_correlations	45	46	1963	3
glass	100	9	529	-	wine	100	13	831	3
heart-statlog-uci	100	13	241	-	wisconsin	100	32	2940	-
hutsof99_child_witness	42	15	296	6					

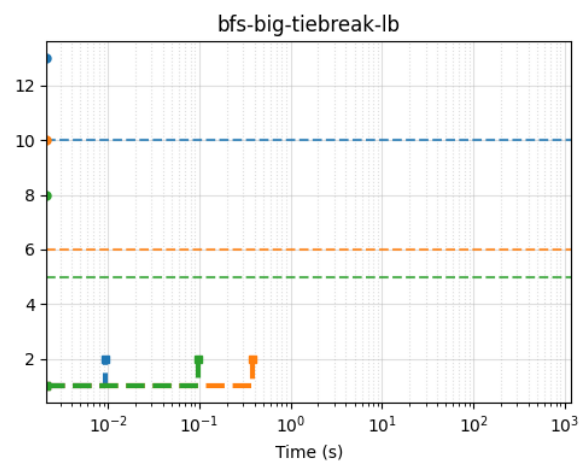
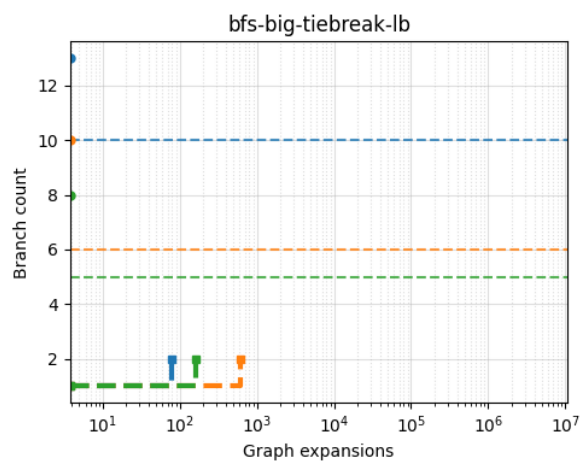
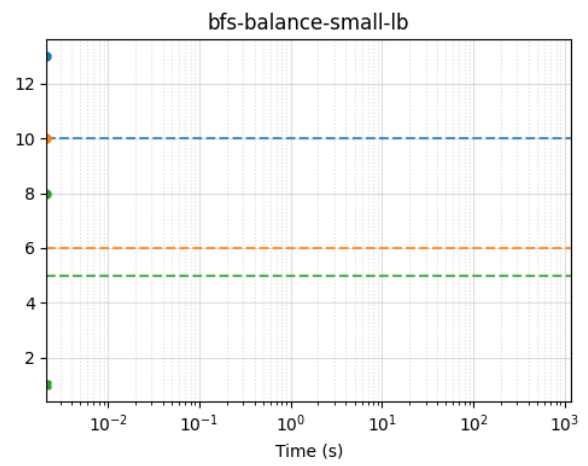
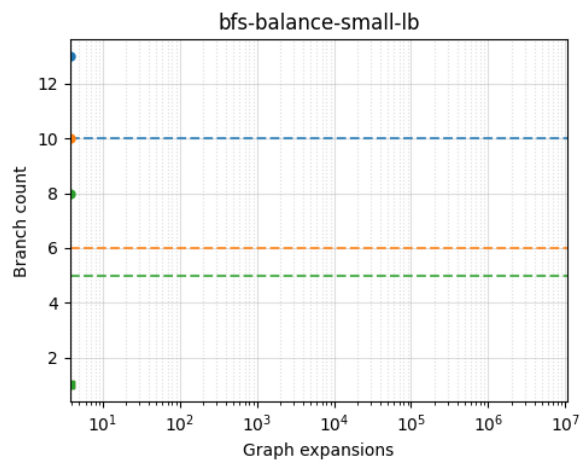
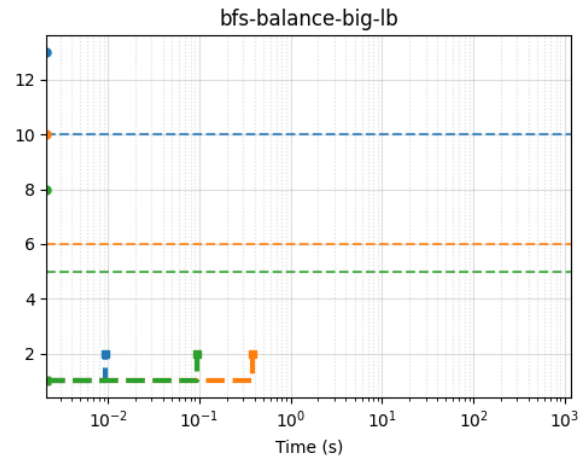
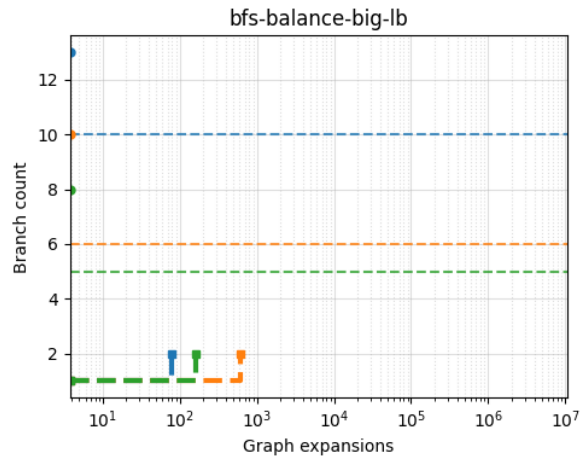
Table A.1: Datasets used in the experiments with the number of examples $\mathcal{D}$, number of features F , the number of feature tests $|W| = \sum_f |W^f|$, and the size of the minimum-size perfect decision tree (- if none of the algorithms found a minimum-size perfect tree).

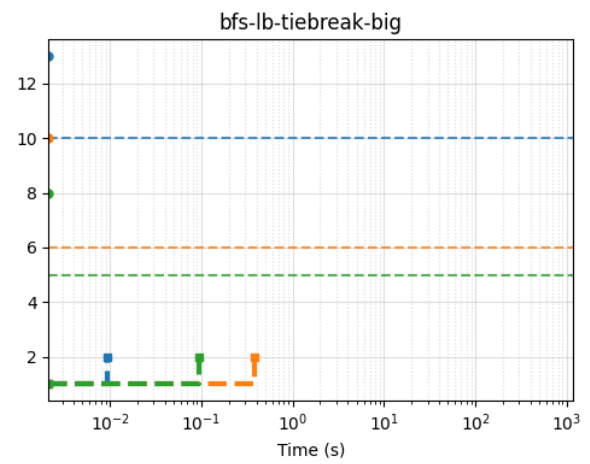
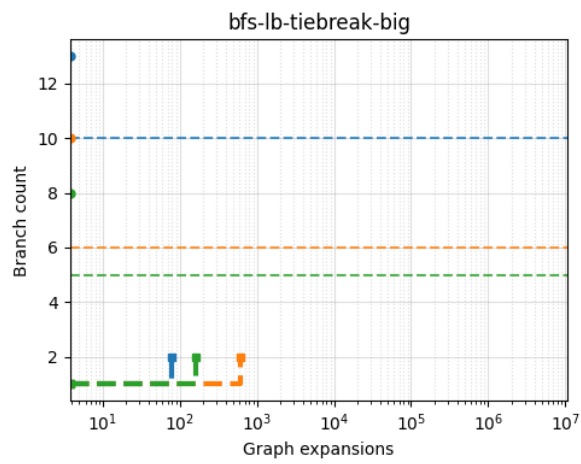
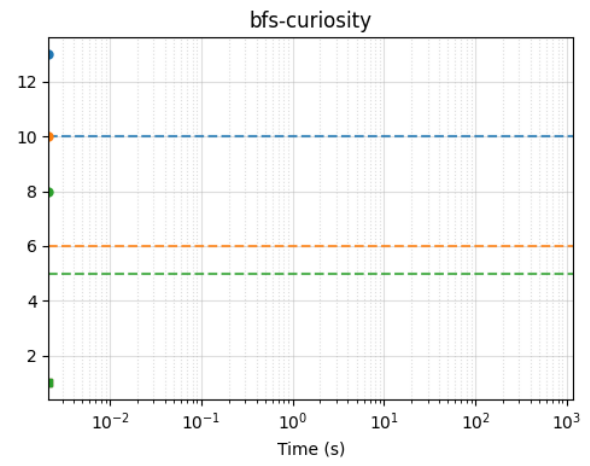
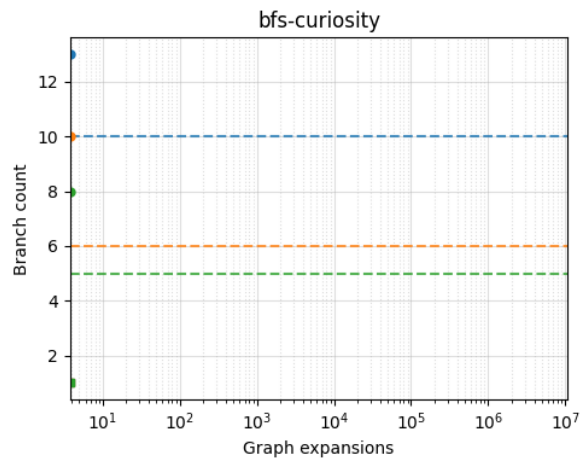
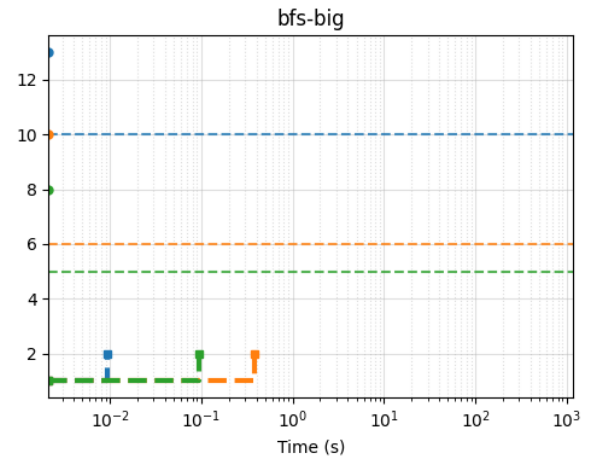
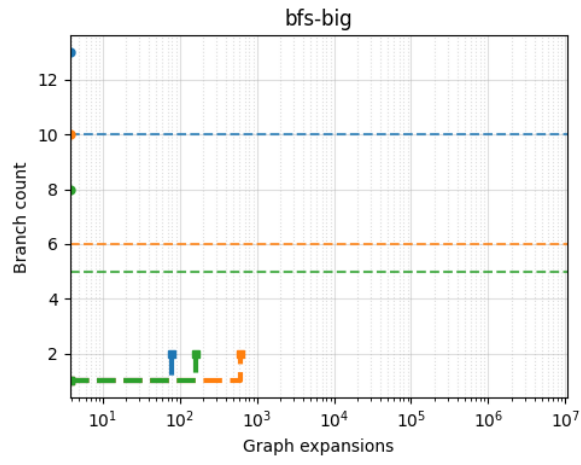
B

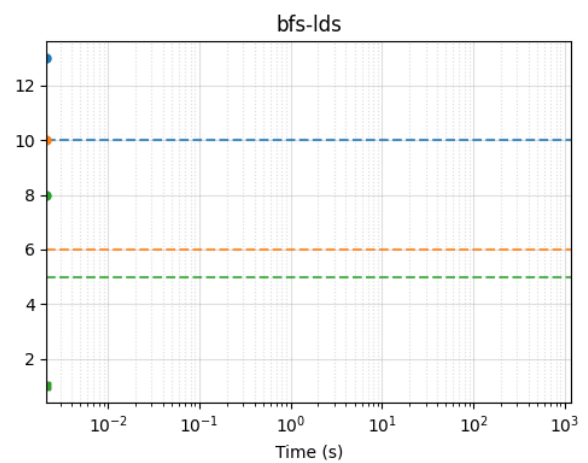
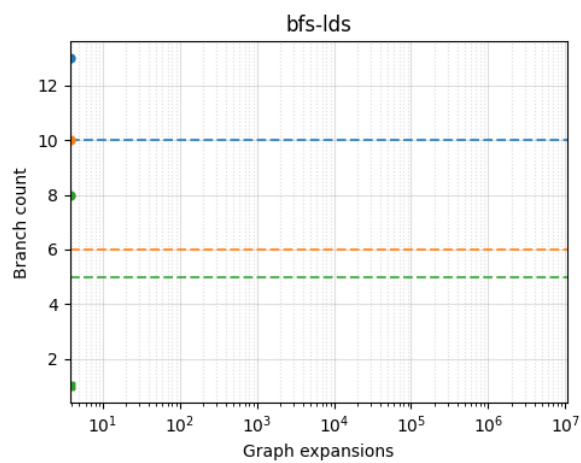
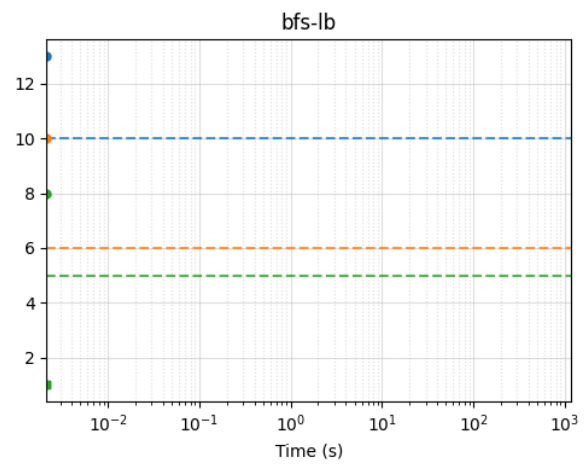
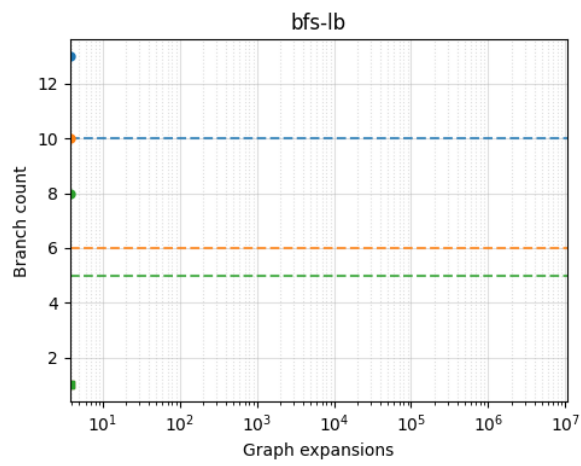
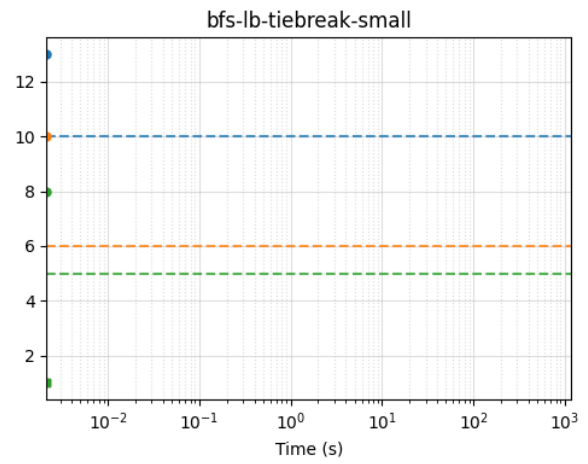
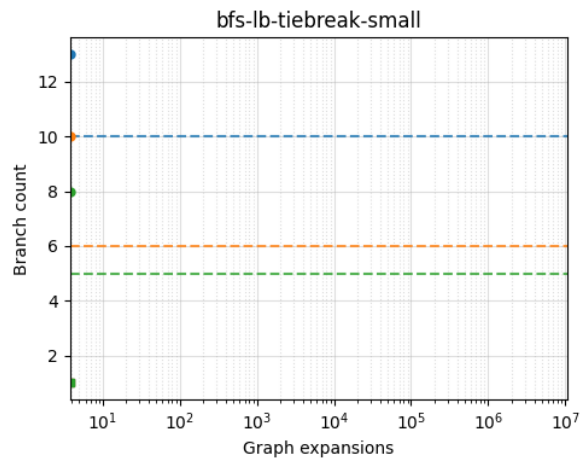
Search Strategies Performance

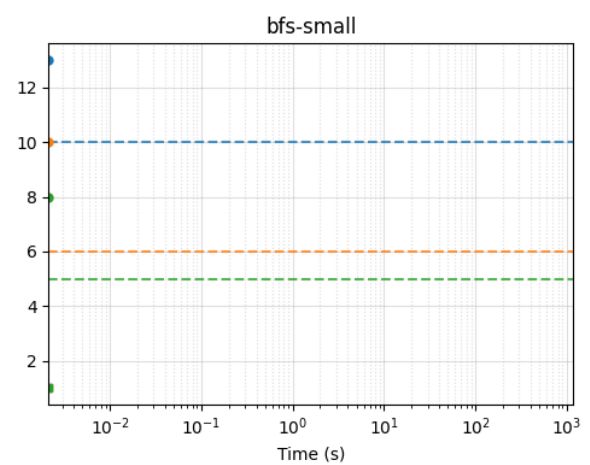
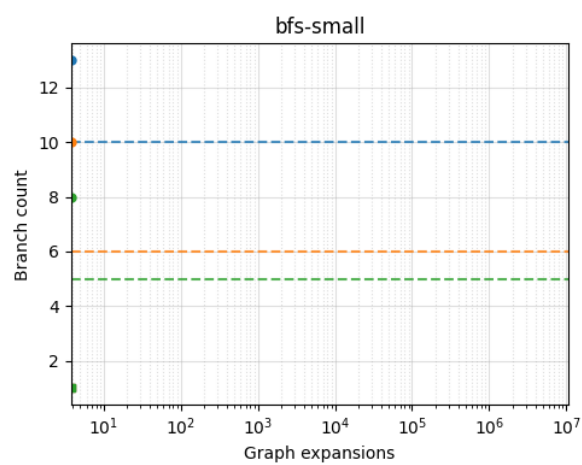
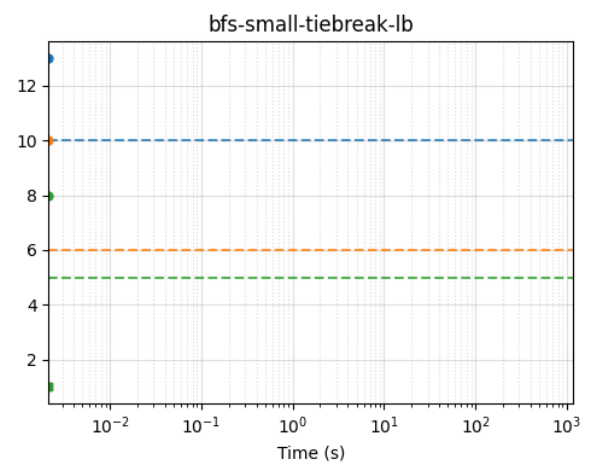
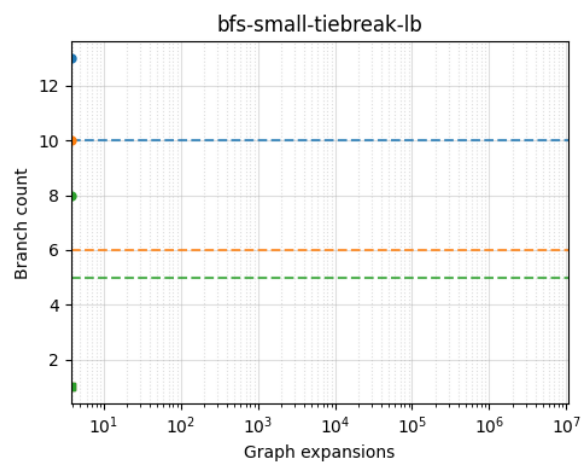
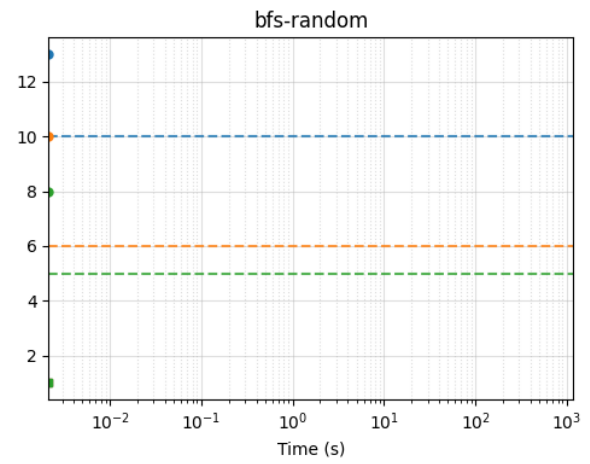
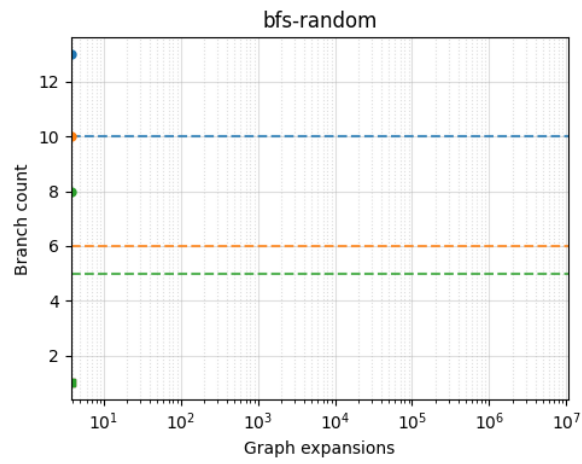
The intermediate lower and upper bounds of all search strategies on the sampled `diggle_table_a1`, `german-credit-data-credibility`, and `pc1` datasets with a time limit of 1200 seconds (20 minutes) and a memory limit of 4GB.

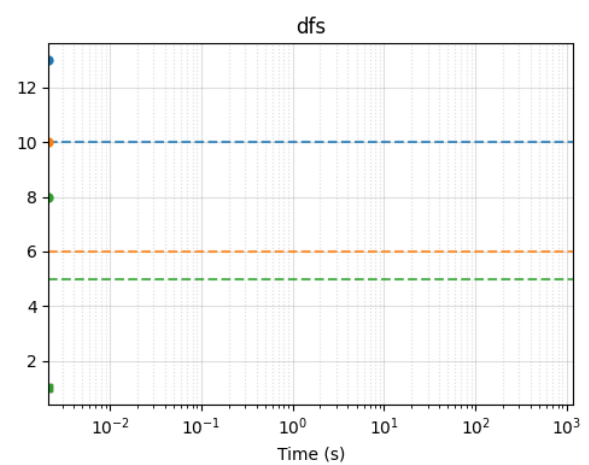
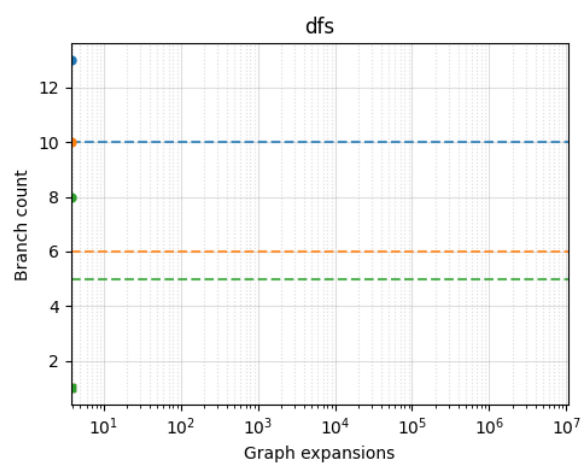
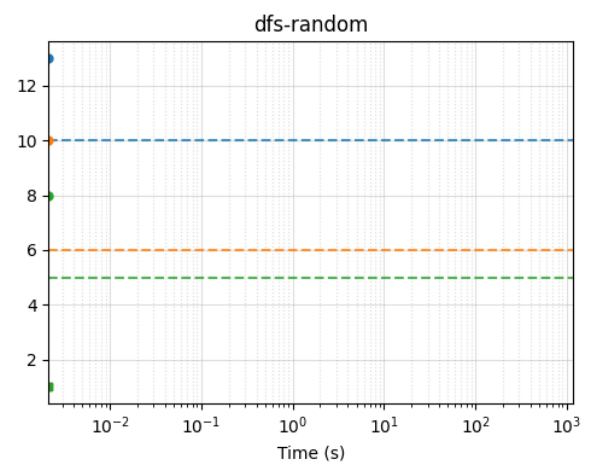
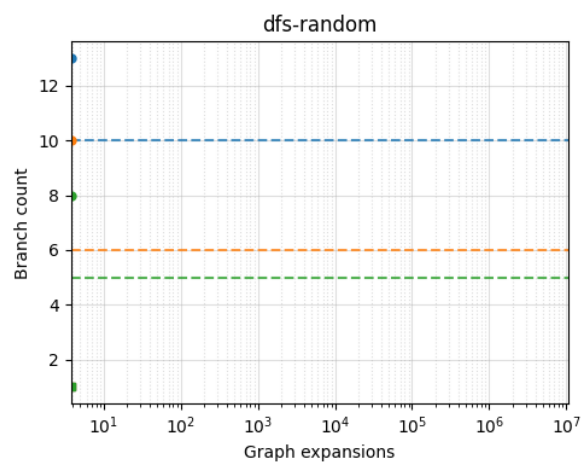
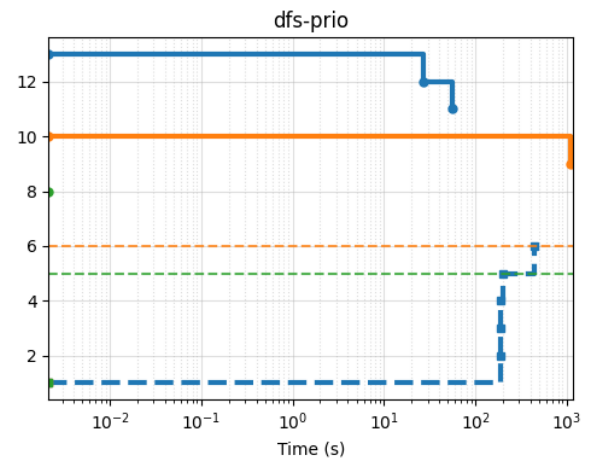
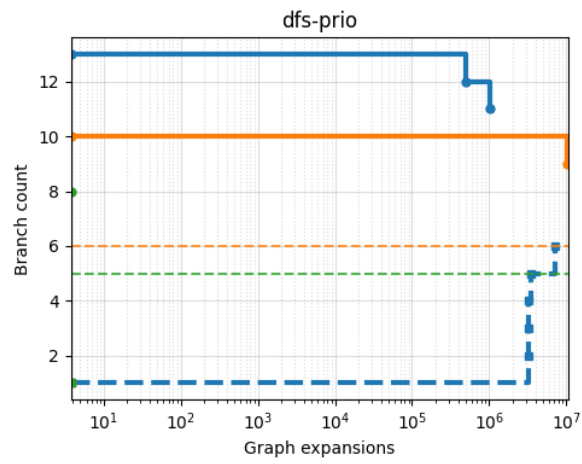












C

Similarity Lower Bound

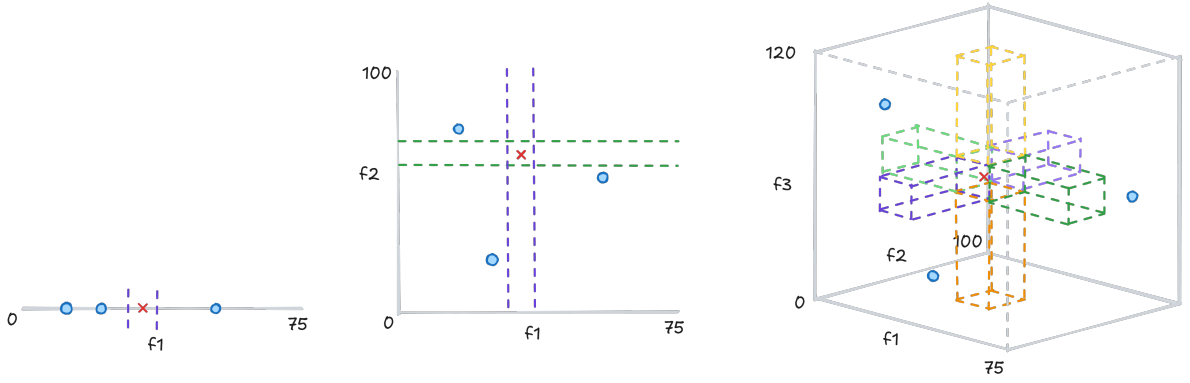


Figure C.1: Axis-aligned isolation of a target point across increasing dimensionality. The red cross denotes the target point while blue circles represent other points. In 1D (left), two bounds on feature f_1 define an interval that isolates the target. In 2D (center), bounds on f_1 and f_2 form an axis-aligned rectangle that excludes the other points. In 3D (right), adding bounds on f_3 produces an axis-aligned box whose intersection of feature ranges encloses only the target point. The dashed regions indicate the allowable ranges along each feature dimension whose intersection isolates the point in feature space.

We explored adapting MurTree’s (Demirović et al. 2022) similarity lower bound to our setting. This bound leverages previously computed lower bounds for a related subset of data to estimate a lower bound for the current subset of data in the search space.

However, the original similarity lower bound was derived in the context of optimal decision tree search with a fixed depth, where the objective is to minimize misclassification. In that setting, removing a single example from the dataset can decrease the misclassification cost by at most 1. In contrast, our setting, MSDT, does not allow misclassifications. Every example must be perfectly separated, so removing examples from a dataset does not correspond to a unit reduction in the objective. Instead, we need to assume that the removed examples were fully isolated.

This leads to a different interpretation of the similarity lower bound. Because the cost of isolating a single example depends on the number of features. As illustrated in Figure C.1, in one dimension, isolating an example requires at most 2 decision nodes. In two dimensions, up to 2 splits per feature may be required, resulting in at most 4 nodes. Extending this to $|F|$ features, isolating a single example requires at most $2|F|$ decision nodes. Consequently, removing $|D_{\text{out}}|$ examples from a dataset D_{old} with lower bound $LB_{D_{\text{old}}}$ can reduce the lower bound by at most $2|F| \cdot |D_{\text{out}}|$, resulting:

$$LB_{D_{\text{new}}}(D_{\text{new}}, D_{\text{old}}, LB_{D_{\text{old}}}) = \max(0, LB_{D_{\text{old}}} - 2|F| \cdot |D_{\text{out}}|) \quad (\text{C.1})$$

Combining this with the class-count lower bound gives:

$$LB_{D_{new}}(D_{new}, D_{old}, LB_{D_{old}}) = \max(CcLB(D_{new}), LB_{D_{old}} - 2|F| \cdot |D_{out}|) \quad (C.2)$$

With $LB_{D_{old}}$ denoting the lower bound computed for the dataset D_{old} , and D_{out} the set of examples removed from D_{old} , i.e., $D_{out} = D_{old} \setminus D_{new}$, the similarity lower bound can be computed using the above formulation. However, the bound decreases by at most $2|F|$ for each removed example, making it increasingly loose as the number of features grows. Consequently, we do not expect this bound to provide substantial pruning on the high-dimensional continuous datasets considered in this work, and therefore omit it from our approach.