AnyDTree: An Anytime Solver for Perfect Decision Trees

Finding progressively smaller trees with 100% training accuracy

Iulia Hosu[1]
Supervisor(s): Emir Demirović[1]**, Koos van der Linden**[1]**, Daniël Vos**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2025

Name of the student: Iulia Hosu
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Koos van der Linden, Daniël Vos, Jasmijn Baaijens

Abstract

Finding the smallest decision tree that perfectly fits the training data is NP-complete; yet, such trees remain attractive due to their interpretability and minimal footprint. Existing solutions occupy two extremes: heuristics like CART instantly produce trees but remain far from optimal, whereas exact solvers like Witty give no intermediate output. We introduce *AnyDTree*, an anytime algorithm that continuously maintains a 100% accurate tree and monotonically shrinks its size. It employs an expand-and-backtrack search that ensures complete solutions at every step, combined with aggressive pruning and caching mechanisms to eliminate redundant exploration. On 70 binary-classification variants of 35 UCI datasets, AnyDTree incurs no statistically significant overhead in finding the optimal size compared with Witty (log-rank $p > 0.1$). On the 46 datasets with known optima, it demonstrates significantly improved anytime behaviour – measured by the confined primal integral – with a median score of 0.00034, outperforming both Witty (0.00059) and CART (0.20) ($p < 0.001$). These results position AnyDTree as a practical middle ground between heuristic and exact solutions.

1 Introduction

Decision trees are a class of machine learning models known for their transparency and ability to capture nonlinear relationships. A **perfect decision tree** achieves 100% accuracy on the training data. Such trees are especially valued when kept small: a tree with fewer nodes is easier to interpret, requires less memory, and yields faster predictions.

In many safety-critical or legally regulated settings, the training table is the specification: every row encodes a condition that *must* be honoured without exception. A conveyor-belt interlock, for instance, must cut power whenever a prescribed pattern of photo-sensors is tripped, and a bedside monitor must raise its alarm exactly when specific and reviewed conditions dictate. In such applications, even a single misclassification could cause injuries or violate regulations, so perfect training accuracy is required. At the same time, the rule set must remain as small as possible: auditors must be able to review the logic manually, and resource-constrained controllers must evaluate it within microseconds.

Unfortunately, finding the smallest decision tree given a dataset is a computationally hard problem – it was proven to be NP-complete by Hyafil and Rivest (1976) [8]. As a result, standard decision tree induction algorithms (e.g. CART [4], C4.5 [12]) rely on greedy splitting heuristics for scalability. These heuristics optimize local criteria (such as information gain); however, they do not guarantee minimal size and typically produce larger or less accurate trees than an optimal solution [15]. This trade-off has long been accepted in practice, where heuristics dominate due to their speed, despite the potential benefits of an optimally small tree.

In recent years, there has been more interest in optimal decision tree algorithms that explicitly attempt to find the smallest or optimal tree for a dataset. Exact formulations using generic solvers have been explored: for example, the problem can be encoded as a SAT [10] or mixed-integer programming [3] model. While these approaches guarantee optimality in theory, they often fail to scale beyond small datasets [5]. Moreover, generic exact solvers are typically not anytime algorithms – they may not produce any feasible decision tree until

the search concludes with an optimal solution. More specialized methods have been produced that exploit the structure of decision tree problems. Dynamic programming (DP) and branch-and-bound techniques can significantly prune the search space [5]. However, even they struggle as the problem size or depth grows, and they generally focus on finding the optimal solution outright rather than producing intermediate results.

A key limitation of existing optimal tree methods is the lack of an anytime property combined with guaranteed perfect accuracy. For instance, the recent Witty solver finds a minimum-size perfect decision tree (the optimal solution) using a specialized *witness tree* search strategy. Still, by design, it does not output any intermediate trees before it finishes the full search [13]. In contrast, Blossom is an anytime algorithm based on dynamic programming that maintains and progressively refines the best tree found during search, without significant extra cost over greedy heuristics [6]. However, Blossom targets a different objective – it maximizes accuracy under a given depth limit, which means it may sacrifice perfect training accuracy to reduce depth. Thus, it does not ensure a *perfect* classifier at all times. Meanwhile, heuristic methods, such as CART, rapidly produce a perfect decision tree using greedy heuristics [4]. This yields a valid solution very quickly, but with no guarantee of optimality. In summary, none of the existing approaches simultaneously guarantees *100% training accuracy* at all times, *anytime output of progressively smaller trees*, and *eventual optimality* in tree size. This clear gap in the literature motivates our work.

In this report, we address the above gap by introducing *AnyDTree*, an anytime algorithm developed and evaluated for binary classification tasks with binarized features. AnyDTree always maintains a *perfect* classifier and monotonically shrinks its size over time. Inspired by the Blossom algorithm [6], AnyDTree employs an expand-and-backtrack strategy: each iteration first *expands* the next unfinished leaf, ensuring the current tree remains complete and valid, then *refines* previously explored branches. A caching mechanism prevents redundant exploration of identical subproblems, while pruning based on computed lower and upper bounds significantly reduces the search space.

On 70 binary-classification variants of 35 UCI datasets, AnyDTree reaches the global optimum without statistically significant overhead in terms of solving time compared to Witty (log-rank $p > 0.1$ in every bucket). On the 46 instances whose optimum is known, it attains a median confined-primal-integral of 0.00034, significantly better than Witty's 0.00059 and CART's 0.20 ($p < 0.001$), thus producing near-optimal perfect trees earlier on average. These results position AnyDTree as a practical middle ground between heuristic methods and exact solvers: it rapidly delivers a solution comparable to CART, progressively improves it over time, and ultimately matches Witty's ability to prove optimality.

2 Related Work

SAT-based and MILP-based exact methods. The optimal decision tree problem has been formulated as a satisfiability or integer programming task. Narodytska et al. (2018) introduced a SAT model for finding the smallest perfect decision tree, encoding the problem with Boolean variables for node decisions [10]. While their SAT approach can ensure optimality, it struggles to handle full datasets directly; the authors had to subsample the training data to make the search feasible.
Similarly, mixed-integer linear programming (MILP) models have been proposed for optimal

decision trees, notably the Optimal Classification Tree (OCT) formulation by Bertsimas & Dunn (2017) [3], and later improvements by Verwer & Zhang (2019) among others [16]. These MILP approaches use integer variables to represent splits and can incorporate regularization on tree size or depth. In practice, however, MILP solvers rarely scale beyond datasets with a few dozen instances or a very limited depth.

Dynamic programming solvers. To overcome the limitations of generic solvers, researchers have developed specialized algorithms that leverage the structure of decision trees. Early work by Nijssen & Fromont (2007) introduced a dynamic programming (DP) approach, DL8, that systematically explores subsets of the dataset to build optimal subtrees [11]. This line of work led to more efficient variants such as DL8.5, which combined DP with branch-and-bound pruning to handle larger search spaces [1]. These methods exploit properties such as optimal substructure, where the optimal tree can be composed of optimal subtrees on splits of the data, and utilize caching to avoid re-solving identical subproblems. Another general framework, STreeD, provides a unifying DP formulation that can optimize various objective criteria, including minimizing misclassifications or tree size, under certain separability conditions [14].

These DP-based algorithms have dramatically improved the scalability of finding an exact decision tree compared to MILP/SAT, and they typically prove optimality by exhaustive search with effective pruning. However, they were generally designed to run to completion and are not inherently anytime – they do not prioritize finding a feasible tree quickly, outputting only the optimal tree at the end of the search.

Witty. Among specialized solvers, Witty is a recent notable contribution designed explicitly for the minimum-size perfect decision tree (MSPDT) problem [13]. Witty implements the witness trees paradigm [9], which efficiently searches the space of decision trees by constructing partial solutions and using them to infer bounds on the final tree size. The authors of Witty report substantial speedups over prior algorithms: on common benchmarks, Witty is an order of magnitude faster than both the MurTree DP solver [7] and earlier SAT-based solutions. This makes Witty one of the current state-of-the-art exact solvers for finding the smallest perfect trees. However, as mentioned, Witty's search is not anytime. It systematically explores the search space and outputs the optimal tree at the end, without intermediate results. If the optimal tree is large or the search space is complex, users must wait until completion to get any solution at all.

Blossom Algorithm. Blossom [6] is an anytime algorithm that adapts a DP-based optimal tree method into an iterative refinement procedure. Instead of targeting perfect accuracy from the start, Blossom incrementally increases tree depth and optimizes accuracy within that depth bound. It can rapidly produce a shallow tree that captures most of the training data and then improve accuracy over time. In doing so, Blossom can output a sub-optimal tree at any time, which is continuously refined. However, because it allows misclassifications when the depth limit is low, Blossom's intermediate trees are not perfect classifiers, and its objective differs from MSPDT – it seeks to maximize accuracy for a fixed depth, rather than minimize size for 100% accuracy.

Heuristic Approaches. Purely heuristic methods remain relevant for quickly obtaining small decision trees. Classical greedy algorithms such as CART and C4.5 can be run without

pruning to yield a perfect-fit tree, then use post-pruning to simplify it at the cost of some training errors. These methods are fast but often produce suboptimal structures – on average, they yield trees that are deeper or less accurate than those found by exact methods [15].

In summary, the literature provides various methods for computing or approximating optimal decision trees. Exact approaches (SAT, MILP, CP, and DP-based) can find the smallest perfect tree but generally suffer from scalability issues and a lack of intermediate results. Newer exact solvers like Witty significantly push the scalability frontier but still do not satisfy the anytime criterion. Blossom can provide faster or iterative solutions, but compromises on perfect accuracy, while heuristic approaches do not provably reach optimality. This research builds on insights from these works to create a new solver that unifies the strengths of both worlds: it guarantees eventual optimality and maintains a perfect classifier at all times, while progressively reducing the tree size in an anytime fashion.

3 Preliminaries

Let $\mathcal{X} \subseteq \{0,1\}^d$ be the feature space and $y : \mathcal{X} \to \{0,1\}$ a binary labeling. A **dataset** $\mathcal{D}$ is a pair (X, y) with $X \subseteq \mathcal{X}$ and $y(x) \in \{0,1\}$ for each $x \in X$. We assume no two examples in $\mathcal{D}$ share identical features with different labels.

A **decision tree** is a full binary tree T in which each internal node is labeled with a cut and each leaf is assigned a class in $\{0,1\}$. Every example $x \in X$ is routed from the root to a unique leaf by testing the cuts on each path. Formally, we can view a tree as a tuple $(V, \mathcal{D}, F, \ell)$ where V is the set of vertices, $F(v)$ is the cut at internal node v, and $\ell(v) \in \{0,1\}$ is the class label at leaf v. We let $|T|$ denote the size of the tree T, defined as the number of internal (decision) nodes (i.e., cuts) in T. At the root r, we assign all examples $X(r) = X$. For an internal node v with cut $F(v) = i$ and left/right children $v_{\mathsf{L}}, v_{\mathsf{R}}$ we set $X(v_{\mathsf{L}}) = \{x \in X(v) : x_i = \text{false}\}, X(v_{\mathsf{R}}) = \{x \in X(v) : x_i = \text{true}\}$, so each example reaches exactly one leaf.

A decision tree T **classifies perfectly** (X, y) (i.e. has zero training error) if for every example $x \in X$, the class $\ell(x)$ of the leaf containing x equals $y(x)$. Equivalently, for each leaf v, the label $\ell(v)$ matches all examples in $X(v)$. In this case, we call T a **perfect** decision tree for (X, y).

Blossom Algorithm. Finally, we recall the *Blossom algorithm* [6] for bounded-depth optimal decision trees. Blossom performs an anytime search that uses two stacks, B and S, to manage partial tree expansions. Informally, incomplete branches (those with non-zero error that have not yet reached the maximum depth k) are called *buds*, and are stored in stack B for further expansion. A refinement stack S records the sequence of splits being explored. At each step, Blossom picks a bud from B, splits it on the next feature, and pushes the choices onto S. When a branch completes or is pruned, Blossom backtracks via S, updating best solutions and possibly reinserting buds into B to try other cuts. The algorithm always grows every unfinished bud before performing refinements; it produces a *full* (intermediate) tree early, and then improves it by testing different splits. This gives it the anytime property, while never re-exploring the same branch and preserving optimality

guarantees. We reuse the same search approach, but change the objective from *minimum error at bounded depth* to *minimum size with zero error* and introduce different pruning and memoization techniques.

4 Methodology

Algorithm 1 outlines the pseudocode of our MSPDT solver, AnyDTree, which employs the same expand-and-backtrack strategy as Blossom. The solver maintains two stacks to manage search states: an expansion stack B for nodes awaiting exploration and a backtracking stack S for their parent nodes. In each iteration of the main loop, the algorithm pops the next node from B to expand it. If this node already represents a solved leaf or cannot lead to a better solution (triggering a pruning condition), it is finalized or pruned without further expansion. Otherwise, AnyDTree selects a feature and splits the node's data into two child nodes. The new left and right children are pushed onto B for subsequent exploration, and the current node is placed onto S so that it can be revisited to try alternative splits later. When the expansion stack becomes empty, the algorithm pops a node from S, prepares it for the next untried feature, and pushes it back onto B – effectively backtracking to explore the next branch. Some details are omitted – such as state caching, dynamic feature ordering, budget updates, and lower-bound calculations for pruning – which are discussed in later subsections.

4.1 Memoization

Many different branches in the decision tree search can lead to the *same subproblem*: classifying an identical subset of examples. For instance, two distinct partial paths might coincidentally filter the dataset down to the same subset $S_b \subseteq \mathcal{D}$. In a brute-force search, the solver would solve this subproblem twice independently. Memoization alleviates this by caching the results of solved subproblems, allowing them to be reused.

AnyDTree maintains a global cache of states, where each state corresponds to a subset of points $X_b \subseteq X$ encountered during the search. For each such subset X_b, we store the best-known subtree (decision tree) that perfectly classifies X_b, including its size and structure. This cache enables effective reuse of partial solutions and facilitates an "upward" propagation of improvements. In particular, whenever the algorithm finds a smaller perfect subtree for some subset X_b, we update the cache record for X_b and then immediately propagate this improvement to any larger subsets that had X_b as part of their split. Each state record contains pointers to its parent states (i.e., references to any superset X_a that was previously split into X_b and its complement $X_c = X_a \setminus X_b$ during search). Using these parent pointers, the solver updates each affected parent's best-known solution: if $|T(X_b)| + |T(X_c)| + 1$ is smaller than the previously recorded size for X_a, it is replaced with the new improved size (and corresponding tree). We then recursively propagate this update further up the chain of parent pointers (so an improvement to X_b can induce an improvement to a parent X_a, then to a grandparent X_d containing X_a, and so on). Ultimately, the global optimum is found in the state for the full training set X. This parent-child state update mechanism ensures that any local discovery of a better subtree immediately benefits all ancestor subsets, avoiding redundant search in those regions of the state space. Furthermore, if we ever prove that a subset's best solution is optimal (for example, when its size equals the calculated lower bound), we mark that state as *optimal*. Marking X_b as solved optimally raises its recorded

lower bound to the optimal value, which in turn can tighten the budget constraints of its sibling, enabling more pruning.

Algorithm 1: Anytime Search for a Minimum-Size Perfect Decision Tree

```
Data: Set 𝒟 containing all data points
Result: Smallest perfect decision tree
root ← 𝒟
B ← { root }
S ← ∅
while |S| + |B| > 0 do
    if |B| > 0 then
        pick and remove b from B
        UpdateBest(b)
        if b is already optimal then
            continue
        if b.domain = ∅ and b.budget ≥ best[b] − 1 or b is pure then
            mark b as optimal; continue
        if b.domain = ∅ then
            continue
        if b.budget < LB(b) then
            b.sibling.budget ← −1
            continue
        pick and remove feature f from b.domain
        split b on f into v_L, v_R
        if (v_L, v_R) are marked in state as children of b with a higher budget then
            B ← B ∪ { b };
            continue
        mark v_L, v_R as children of b with budget b.budget in state;
        if v_L.budget ≥ LB(v_L) and v_R.budget ≥ LB(v_R) then
            B ← B ∪ {v_L, v_R}
        else
            B ← B ∪ { b }
    else
        while |S| > 0 do
            pick and remove b from S
            UpdateBest(b)
            if b is pure or optimal and LB(b) = best[b] then
                mark b as optimal; continue
            if b.domain ≠ ∅ then
                B ← B ∪ { b }
                break
```

Although checking the cache and storing entries can take $O(n)$ worst-case time, the overhead is justified as every subsequent cache hit replaces what would otherwise be an exponential sub-search. The larger cost comes from memory: in the worst case, the search might encounter a distinct memoization key for every *subset of points or* for every *partial feature assignment*. There are at most 2^n possible distinct subsets of points and 3^m distinct partial feature assignments, because each of the n examples can be either present or absent in a subset, while along a root-to-bud path, each feature appears in three possible states: *unused*, *queried and evaluated to 0*, or *queried and evaluated to 1*. Hence, the total number of unique states in the cache is bounded by $\min(2^n, 3^m)$. To manage memory, we impose a *cache*

cap: once the number of states stored in the cache exceeds a given threshold (specified as a parameter), all states are deleted, except those that are part of the current tree. This means that some states might be explored multiple times, resulting in additional time overhead; therefore, a larger cache limit is usually preferred, depending on the amount of available memory.

4.2 Lower Bound Estimate

For every search node v, we reuse the *Improvement Lower Bound* (ImpLB) of the WITTY solver [13]. Write $X(v) \subseteq X$ for the examples reaching v, let $\text{maj}(v) = \max_{c\in\{0,1\}} |\{x \in X(v) \mid y(x) = c\}|$ be the majority class, and denote the misclassified points by

$$M(v) = \big\{x \in X(v) \mid y(x) \neq \text{maj}(v)\big\}$$

Splitting v on feature $i \in \{1, \ldots, d\}$ partitions the data into $X(v_\mathsf{L})$ and $X(v_\mathsf{R})$. When each child leaf is labeled with its majority, feature i *repairs* errors:

$$\text{fix}(i, v) = \big|\{x \in M(v) \mid y(x) = \text{maj}(v_\mathsf{L}) \text{ if } x_i = 0,\ y(x) = \text{maj}(v_\mathsf{R}) \text{ if } x_i = 1\}\big|$$

Define the set-cover instance $\mathcal{C}(v) = \langle U, F\rangle$ with universe $U = M(v)$ and family $\text{F} = \{\text{imp}(i, v)\}_{i=1}^{d}$, where $\text{imp}(i, v)$ is the set counted by $\text{fix}(i, v)$. Every perfect decision tree T for (X, y) selects at most one feature per internal node; the $|T|$ selected improvement sets cover *all* mistakes at every node on every root-to-leaf path, hence form a cover of size $\leq |T|$ for $\mathcal{C}(v)$. Consequently, any lower bound for this set-cover instance is also a lower bound for the optimal tree size below v.

To calculate the lower bound, we sort the values $\text{fix}(i, v)$ in non-increasing order and add features until their cumulative sum meets $|M(S)|$:

$$\text{ImpLB}(v) = \min\Big\{t \;\Big|\; \sum_{j=1}^{t} \text{fix}(i_j, v) \geq |M(S)|\Big\}$$

Computing $\text{fix}(i, v)$ scans $d \cdot |X(v)|$ feature bits and the final sort costs $O(d \cdot \log d)$, resulting in $O(d \cdot |X(v)| + d \cdot \log d)$ per call. We memoize each value in the hash table `LB`; an entry is removed as soon as $X(v)$ is deleted from the main memoization table `best`, keeping the cache size bounded by the selected cache cap.

4.3 Branch-and-Bound

To guarantee optimality, our algorithm systematically explores the search space, but we prune branches that cannot lead to a better (smaller) solution than the best one already found. We employ a *branch-and-bound* scheme with a key component: maintaining a *budget* of remaining nodes for each branch.

4.3.1 Budget Propagation

The *budget* of a branch b is the maximum number of splits (internal nodes) its subtree may use while still having a chance to beat the current best solution. Let U be the size of the best

perfect tree found so far. For the root, we therefore start with budget $B = U - 1$ because we are only interested in trees that are *strictly* smaller than the incumbent. When we expand a branch into two children, the branch itself will use one node for the split, leaving a budget of $B - 1$ to distribute among the two child subtrees. We propagate the budget to the children by subtracting the cost of the split and reserving nodes for the other child based on its lower bound. Concretely, assume a branch with budget B is split on feature f, producing children b_L and b_R. The split itself consumes one internal node, so only $B - 1$ nodes remain for both sub-trees. Each child must also leave at least $LB(\text{sibling})$ nodes for the other child. Moreover, if a smaller perfect subtree has already been discovered for a child's subset, we can safely tighten its budget to that known value. Hence we set

$$\boxed{\begin{aligned} B_L &= \min\big(\text{best}_L,\ B - 1 - \text{LB}(b_R)\big), \\ B_R &= \min\big(\text{best}_R,\ B - 1 - \text{LB}(b_L)\big), \end{aligned}}$$

where $\text{LB}(\cdot)$ is the lower bound for a subset and $\text{best}_{L/R}$ is the size of the best perfect subtree already found for that subset (if any). If either budget becomes infeasible, i.e.

$$B_L < \text{LB}(b_L) \quad \text{or} \quad B_R < \text{LB}(b_R),$$

then no completion of the split can beat the current best tree, and the split on f is pruned. These tighter budgets never eliminate an optimal solution; they are derived solely from proven lower bounds or from already found feasible subtrees.

4.3.2 Additional Pruning Rules

Two further budget-based tests (lines 10 and 19 of Algorithm 1) tighten the search without compromising optimality. First, when a branch b has *no* untried features left ($\text{domain}(b) = \varnothing$), it is declared *optimal if* $b.\text{budget} \geq \textit{best}[b] - 1$. The rationale is that if the budget were any smaller, a perfect tree of size $\textit{best}[b] - 1$ might still lie undiscovered, so the branch cannot be marked as optimal (fully explored).

Second, when the solver proposes to split b into children (v_L, v_R), the split is *skipped* if the same pair has been encountered before with a *larger* budget ($\text{budget}_{\text{old}} > \text{budget}_{\text{new}}$): every subtree admissible under the smaller budget was already considered in the earlier, more permissive search. Conversely, if (v_L, v_R) was seen only with a *smaller* budget, the split *must* be re-explored, because a solution of size $\text{budget}_{\text{old}} < |T| \leq \text{budget}_{\text{new}}$ could have been missed. These two tests are inexpensive and yet eliminate a large number of redundant expansions in practice.

4.4 Heuristic Feature Ordering

We employ a heuristic ordering of features to guide the search at each branch. The idea is to use a simple impurity measure on the dataset to rank features by their usefulness for splitting, and then use this fixed order throughout the search. We choose Gini impurity as our ranking criterion, which is commonly used in decision tree learning. During the search, whenever a branch needs to choose the next feature to test, it will select features in this predetermined order. As reordering features for one subset is $O(m \cdot |S_b|)$ per node, doing this at every branch in an exhaustive search is costly; thus, except for the first full tree expansion, we use a global order. We reduce the branching factor early on by considering

the most promising splits first, leading to a better solution being found in the initial stage, thereby tightening the upper bound and enabling more pruning in subsequent searches. We perform this computation once in $O(m \cdot n)$ time for each node in the first expanded tree. This is a negligible overhead overall, given that the search itself is exponential.

5 Experimental Setup and Results

We evaluate the proposed algorithm from two perspectives. First, we study its *anytime behaviour*, that is, how quickly and how far it reduces the size of a perfect tree while the search is still running. The one-number summary we adopt is the *Confined Primal Integral* (CPI) [2]: a lower CPI rewards algorithms that deliver small trees early and continue to improve them over time. Second, we analyse the AnyDTree's ability to *reach and certify the global optimum*. For this, we consider each dataset as a "time-to-event" observation (the event being the finding or proving of optimality) and compare the survival curves using log-rank tests.

Throughout the experiments, we compare AnyDTree with two established baselines: CART [4], a widely used heuristic algorithm for finding decision trees, and Witty [13], a state-of-the-art exact solver designed to find minimum-size perfect decision trees. CART is included only in the anytime analysis, as it finds a tree instantly but never proves optimality, whereas Witty serves as a reference in both evaluations.

On the 46 benchmark instances for which the global optimum is known, AnyDTree achieves a median CPI of 0.00034, significantly better than Witty's 0.00059 and CART's 0.20 ($p < 0.001$). In the time-to-optimality study, AnyDTree and Witty solve a similar number of datasets across all categories; log-rank tests reveal no statistically significant difference between their survival distributions ($p > 0.1$ in every subgroup). In short, AnyDTree exhibits improved anytime behaviour, while showing no statistically significant differences from Witty in terms of reaching optimality.

5.1 Experimental Setup

All experiments were performed on a desktop system configured with Windows 11 Pro 24H2, equipped with 32 GB of RAM and powered by an AMD Ryzen 7 8700G CPU with 8 cores, 4.2 GHz. To ensure controlled conditions, each experiment was allocated one logical CPU core, allowing for up to four experiments to be executed concurrently. A strict one-hour timeout was enforced for each experiment.
AnyDTree's experiments were run with a cache limit of 2,000,000 entries. This cache size was determined empirically to strike a balance between speed and memory efficiency, consistently maintaining memory usage below 8 GB per run, with typical usage observed at under 4 GB. Witty experiments were executed without an explicit upper-bound on tree size, with its limit set arbitrarily high at 9999.

5.2 Datasets

We utilized the dataset collection provided in the Witty archive,[1] selecting the 35 UCI datasets preprocessed for binary classification. The preprocessing involved one-hot encod-

[1] `https://zenodo.org/records/14855274`

ing of categorical attributes, thresholding of numerical attributes, and merging of minority classes into a single class, resulting in binary classification tasks. Conflicting duplicate rows were removed. The Witty archive provides ten random samples of each dataset at both 20% and 50% sizes [13]. To keep our evaluation concise, we selected the first sample from each size, resulting in a total of 70 datasets. The selected datasets have a median feature count of 189 and a median instance count of 68. Detailed characteristics of these datasets, including the number of features and instances, are provided in the paper's Appendix A.

5.3 Anytime Behaviour

The anytime nature of AnyDTree is evaluated by examining how the relative tree size evolves over time and by quantifying the overall performance using the *Confined Primal Integral* (CPI) metric [2]. The results are normalized by the optimal size for each dataset; thus, these analyses focus on the 46 out of 70 datasets for which an optimal tree size was established by either AnyDTree or Witty within the one-hour time limit.

5.3.1 Mean Anytime Performance

Figure 1 depicts, on a logarithmic time axis, the *mean* size (internal node count) of the smallest perfect decision tree found so far, expressed as a ratio to the proven optimum (1.0 = optimal). The plot aggregates the 46 datasets on which the optimum is known.
CART produces its tree almost instantaneously and never changes it, so the blue line remains flat at about 1.31, roughly 30% larger than optimal. AnyDTree returns a first tree roughly just as quickly, then keeps shrinking it, reaching ≈ 1.18 at ten seconds, and ≈ 1.02 by the one-hour time limit. Witty publishes no tree before it has completed its proof, so until that moment, we consider its size equal to CART's solution. In the mean curve, this behaviour appears as the orange line, which starts at the CART level and then descends gradually as different datasets are solved at different times.
Taken together, the curves show that AnyDTree quickly provides trees that are smaller than CART's and also smaller than the best mean tree size available from Witty.
Notably, a limitation of these figures is that they average performance over a subset of datasets where optima were found, potentially not reflecting behaviour on the most computationally challenging instances.

5.3.2 Confined Primal Integral

The CPI compresses an anytime-performance curve into a single number that rewards both early progress and eventual solution quality [2]. For a minimisation problem and a time limit T it is defined as

$$\mathrm{CPI}(T) \;=\; \frac{1}{T}\int_0^T p(t)\, e^{\,t/\alpha}\, dt, \qquad \alpha \;=\; \frac{T}{\ln \iota},$$

where $\iota \in (0,1)$ is an *importance factor* that controls the decay of the exponential weight. The primal-gap function $p(t)$ is

$$p(t) = \begin{cases} 1, & \text{if no feasible tree has been found by time } t, \\ \dfrac{size_t - size_{optimal}}{size_t}, & \text{otherwise,} \end{cases}$$

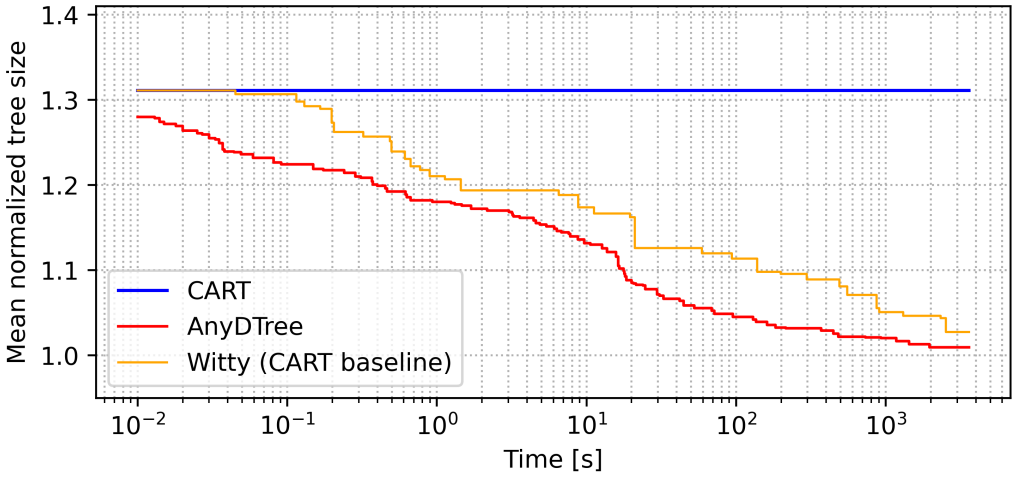

Figure 1: Mean normalised tree size over time (log scale) on the 46 datasets whose optimum is known. For each dataset, we consider Witty's solution equal to CART until Witty finishes. Lower is better.

with $size_t$ the size of the incumbent tree at time t and $size_{optimal}$ the *globally optimal* tree size. Because our benchmark contains many instances whose optimum is proven, we restrict the CPI evaluation to the 46 datasets for which $size_{optimal}$ is known and substitute that exact value in the equation.

Throughout the experiments, we use $\iota = 0.10$. For normalization, the raw integral is divided by its worst-case value (gap $= 1$ for the whole run), resulting in a range of $[0, 1]$; a CPI of 0 indicates the optimal tree was found instantly, while a CPI of 1 indicates the primal gap never decreases during the run.

Table 1 lists the **median** CPI across the 46 fully solved instances and shows Wilcoxon signed-rank tests against AnyDTree. AnyDTree obtains the lowest median CPI, 0.00034, and significantly outperforms both baselines ($p < 0.001$ in each comparison). The full CPI results for each dataset are presented in Appendix C.

Algorithm	Median CPI	Wilcoxon W (vs. AnyDTree)	p-value
CART	0.20000	0.0	<0.001
Witty	0.00059	194.0	<0.001
AnyDTree	**0.00034**	—	—

Table 1: Median confined primal integral (lower is better) and Wilcoxon signed-rank tests versus AnyDTree on the 46 datasets with proven optimal tree size ($\iota = 0.10$, normalized).

5.4 Reaching Optimality

Solved Datasets. Figure 2 illustrates the cumulative percentage of all 70 datasets for which Witty and AnyDTree achieve optimality over time, plotted on a logarithmic scale.

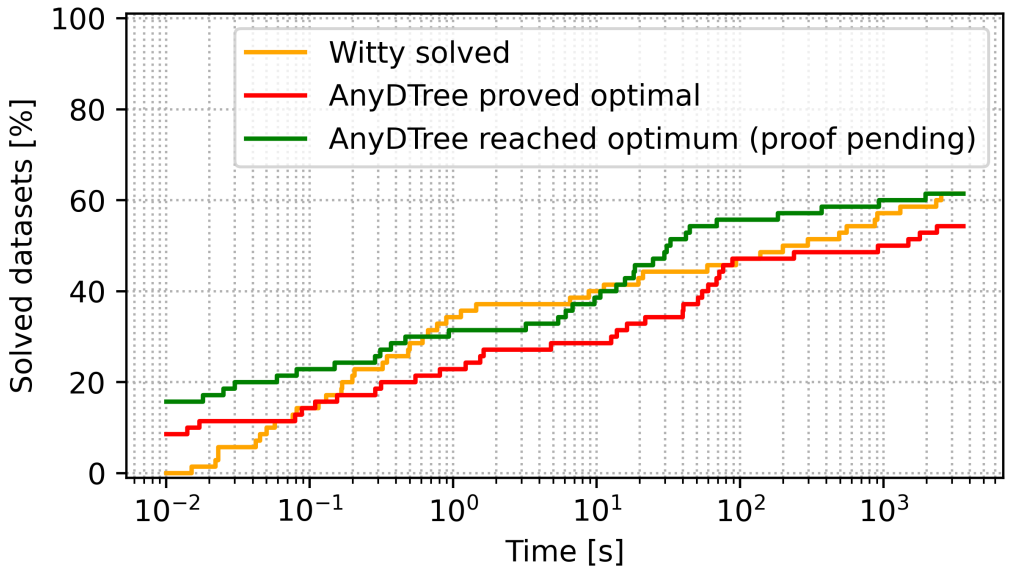

Figure 2: Cumulative percentage of the 70 datasets solved to optimality over time (log scale). 'Witty solved': Witty proves optimality. 'AnyDTree proved optimal': AnyDTree finishes and proves optimality. 'AnyDTree reached optimum (proof pending)': AnyDTree finds a tree of a size that is later confirmed to be optimal (either by the AnyDTree's proof or by Witty's proof); if neither algorithm proves optimality for a dataset, that dataset does not contribute to this curve, as the optimum is unconfirmed. Higher is better.

The orange curve ("Witty solved") shows that Witty proved optimality for approximately 61.4% (43 out of 70) of the datasets within the one-hour timeout. The red curve ("AnyDTree proved optimal") indicates that the AnyDTree proved the optimality of its found tree for about 54.3% (38 out of 70) of the datasets. The green curve ("AnyDTree reached optimum (proof pending)") tracks instances where AnyDTree found a tree of a size that was later confirmed to be optimal, either by AnyDTree's own proof or by Witty's. This curve reaches approximately 61.4% (43 out of 70 datasets), matching Witty's final solved count.

Survival analysis. Kaplan-Meier curves were generated for each solver in four buckets (many instances / few instances and many features / few features) and can be found in Appendix D. The threshold for splitting is 68 for instances and 189 for features, representing the medians across all used datasets. Log-rank tests (Table 2) show no significant difference in time-to-proof between Witty and the AnyDTree in any bucket (all $p > 0.1$). The high-feature group exhibits a non-significant trend favouring Witty. In contrast, AnyDTree's time to find the optimal solution is marginally faster on low-feature datasets, but neither effect reaches statistical significance at $\alpha = 0.05$. We therefore conclude that, once time-outs are treated as censored observations, the two solvers have comparable overall finish-time performance across the benchmark.

		Solved datasets in ≤1 h			AnyDTree proved vs. Witty		AnyDTree found vs. Witty	
Bucket	n	Witty	AnyDTree proved	AnyDTree found	χ^2	p	χ^2	p
All datasets	70	43	38	43	0.72	0.40	0.08	0.77
$\|F^*\| \leq 189$	36	29	29	31	0.00	0.96	1.60	0.21
$\|F^*\| \geq 189$	36	16	10	14	2.72	0.10	0.29	0.59
$\|D\| \leq 68$	36	32	30	32	0.39	0.53	0.45	0.50
$\|D\| \geq 68$	36	11	8	11	0.69	0.41	0.00	0.99

Table 2: Solved instances within 1h for each bucket (Witty, AnyDTree at proof, AnyDTree at first hit) and the log-rank statistic $\mathcal{X}^2/p$ comparing AnyDTree with Witty; no bucket shows a significant difference at p<0.05.

6 Responsible Research

This section summarizes the steps we took to ensure that the research was conducted in a manner consistent with current best practices for responsible research. We discuss ethical considerations surrounding data use and potential bias, the broader societal impact of our contributions, and the concrete measures we have adopted to maximize transparency and reproducibility.

Ethical Considerations. All empirical tests in this paper rely solely on *publicly available benchmark data sets*. No personal, proprietary, or sensitive data was used, so the experiments pose minimal privacy risk. Nevertheless, AnyDTree is a *model-induction* technique: it will reproduce *any statistical bias present in the training data*. If the data encodes any bias, the resulting tree can replicate or even amplify it. We therefore urge users to perform standard fairness verifications on the model before deployment.

Societal Impact. A solver that constructs the *smallest* perfect decision tree can advance interpretability and compression in several high-stakes domains, such as medical diagnosis, credit scoring, and others. Smaller trees are easier for human experts to validate and can reduce inference latency on low-resource hardware. Although we evaluated AnyDTree only on standard public benchmarks, its behaviour on unseen, domain-specific data has not yet been studied. Practitioners should therefore validate the method using their own data – and, where relevant, follow any domain-specific regulations – before using it in production.

Transparency and Reproducibility. All artefacts needed to verify our findings are publicly available. The GitLab repository[2] contains the complete source code, the benchmark datasets, and step-by-step instructions for reproducing every experiment. It also documents the exact software environment and command lines we used to generate the results reported in this paper.

7 Conclusions and Future Work

This work bridges the gap between fast – but sub-optimal – heuristics and exact solvers that lack anytime behaviour. The proposed solver outputs a perfect decision tree within millisec-

[2] `https://gitlab.tudelft.nl/brp-25q4/anytime_perfect_trees`

onds, improves it as time allows, and ultimately reaches the optimum. It *finds* the optimal tree on 61.4% of the datasets and *proves* optimality on 54.3% within the one-hour time limit. Across 46 cases with known optima, its CPI is significantly lower than CART's and Witty's, confirming superior anytime behaviour ($p \leq 0.001$). Survival analysis reveals no statistically significant difference in time-to-proof between AnyDTree and Witty ($p \geq 0.1$), indicating that AnyDTree's anytime behaviour does not come with a significant performance cost. These properties make the proposed solver a practical choice when an interpretable, compact model is required but the time budget is too tight for conventional exact methods.

Looking ahead, three directions stand out. First, a multi-threaded implementation would enable the search to explore different branches in parallel, making better use of modern multi-core and GPU hardware. Second, extending the method to handle multi-class classification problems and continuous features would broaden its practical scope beyond the current binary setting. Third, designing tighter theoretical lower bounds on tree size would prune the search space more aggressively. Pursuing these improvements should further increase the solver's speed and versatility while preserving its anytime behaviour and exact guarantees.

A Dataset Shapes

Dataset	$\|F\|$	20% sample		50% sample	
		$\|D\|$	$\|F^*\|$	$\|D\|$	$\|F^*\|$
appendicitis	523	21	89	53	264
australian	1155	138	429	345	788
auto	961	40	357	101	629
backache	469	36	176	90	320
biomed	735	41	211	104	450
breast-cancer	40	53	34	133	40
bupa	307	68	168	170	238
cars	704	78	250	196	447
cleve	390	60	189	151	301
cleveland	391	60	182	151	291
cleveland-nominal	17	26	15	65	17
cloud	585	21	106	54	290
colic	408	71	234	178	327
contraceptive	66	271	62	679	64
dermatology	188	73	161	183	180
diabetes	1246	153	509	384	875
ecoli	351	65	189	163	297
glass	894	40	248	102	539
glass2	709	32	190	81	422
haberman	89	56	56	141	79
hayes-roth	15	16	15	42	15
heart-c	390	60	196	151	308
heart-h	325	58	140	146	234
heart-statlog	376	54	179	135	292
hepatitis	355	31	135	77	245
hungarian	325	58	140	146	241
lupus	126	17	23	43	55
lymphography	50	29	46	74	50
molecular_biology_promoters	228	21	226	53	228
new-thyroid	329	43	130	107	234
postoperative-patient-data	22	14	19	36	21
schizo	2218	68	599	170	1340
soybean	133	124	104	311	106
spect	22	43	22	109	22
tae	96	21	44	53	71

Table 3: Size statistics for the Witty benchmark: number of binary features $|F|$, number of non-redundant binary features $|F^*|$, and number of examples $|D|$ for the 20 % and 50 % subsamples of each base dataset.

B Results

Dataset	Witty Size	Witty Time	AnyDTree Size	AnyDTree Time	AnyDTree Optimality Time	CART Size
appendicitis	3	<1s	3	<1s	<1s	3
australian	-1	timeout	16	137s	timeout	20
auto	4	<1s	4	45s	72s	7
backache	3	<1s	3	<1s	<1s	3
biomed	5	<1s	5	<1s	76s	7
breast-cancer	10	20s	10	3s	5s	12
bupa	-1	timeout	16	1475s	timeout	20
cars	6	11s	6	33s	timeout	8
cleve	8	490s	8	6s	timeout	11
cleveland	7	138s	7	183s	1790s	12
cleveland-nominal	7	<1s	7	<1s	<1s	7
cloud	5	<1s	5	<1s	2s	5
colic	-1	timeout	9	25s	timeout	12
contraceptive	-1	timeout	61	959s	timeout	79
dermatology	1	<1s	1	<1s	<1s	1
diabetes	-1	timeout	23	4s	timeout	26
ecoli	4	<1s	4	<1s	14s	6
glass	5	1s	5	18s	88s	8
glass2	5	<1s	5	14s	40s	7
haberman	13	553s	13	11s	13s	19
hayes-roth	2	<1s	2	<1s	<1s	2
heart-c	7	59s	7	18s	timeout	9
heart-h	-1	timeout	10	30s	timeout	13
heart-statlog	7	94s	7	69s	912s	9
hepatitis	4	<1s	4	<1s	<1s	5
hungarian	8	2522s	8	370s	2354s	14
lupus	7	<1s	7	<1s	<1s	7
lymphography	5	<1s	5	<1s	<1s	7
molecular_biology_promoters	2	<1s	2	<1s	<1s	2
new-thyroid	2	<1s	2	<1s	<1s	2
postoperative-patient-data	4	<1s	4	<1s	<1s	4
schizo	-1	timeout	11	2009s	timeout	12
soybean	6	1s	6	<1s	60s	7
spect	7	<1s	7	<1s	<1s	8
tae	5	<1s	5	<1s	<1s	6

Table 4: Raw results for the 20 % sampled Witty benchmark. "AnyDTree Optimality Time" refers to the additional time required for our solver to verify optimality after determining the best size so far ("AnyDTree Size"). "timeout" indicates that the method reached the 1h cap without completing or proving optimality. "-1" means no feasible tree was found within the time limit.

Dataset	Witty Size	Witty Time	AnyDTree Size	AnyDTree Time	AnyDTree Optimality Time	CART Size
appendicitis	4	<1s	4	30s	40s	5
australian	-1	timeout	43	23s	timeout	47
auto	-1	timeout	12	984s	timeout	16
backache	-1	timeout	10	5s	54s	13
biomed	6	21s	6	1960s	timeout	16
breast-cancer	-1	timeout	29	372s	timeout	42
bupa	-1	timeout	42	3333s	timeout	46
cars	-1	timeout	14	76s	timeout	16
cleve	-1	timeout	20	115s	timeout	24
cleveland	-1	timeout	22	105s	timeout	24
cleveland-nominal	15	1307s	15	<1s	<1s	18
cloud	8	2326s	9	<1s	timeout	9
colic	-1	timeout	22	<1s	timeout	23
contraceptive	-1	timeout	198	1687s	timeout	231
dermatology	3	<1s	3	<1s	1s	3
diabetes	-1	timeout	62	3121s	timeout	69
ecoli	6	9s	7	2s	timeout	10
glass	-1	timeout	18	4s	timeout	23
glass2	-1	timeout	14	225s	timeout	19
haberman	-1	timeout	28	928s	1488s	40
hayes-roth	9	<1s	9	<1s	<1s	12
heart-c	-1	timeout	21	<1s	timeout	24
heart-h	-1	timeout	24	127s	timeout	25
heart-statlog	-1	timeout	19	2104s	timeout	23
hepatitis	7	868s	8	53s	timeout	12
hungarian	-1	timeout	19	2s	timeout	21
lupus	12	<1s	12	<1s	2s	15
lymphography	10	296s	10	10s	68s	13
molecular_biology_promoters	4	7s	4	16s	22s	5
new-thyroid	5	<1s	5	25s	51s	6
postoperative-patient-data	9	<1s	9	<1s	<1s	14
schizo	-1	timeout	33	854s	timeout	38
soybean	9	198s	9	42s	timeout	10
spect	14	904s	14	7s	16s	17
tae	-1	timeout	15	31s	237s	23

Table 5: Raw results for the 50 % sampled Witty benchmark. "AnyDTree Optimality Time" refers to the additional time required for our solver to verify optimality after determining the best size so far ("AnyDTree Size"). "timeout" indicates that the method reached the 1h cap without completing or proving optimality. "-1" means no feasible tree was found within the time limit.

C Confined Primal Integral

Dataset	Witty	AnyDTree	CART
appendicitis 20%	0.0	0.0	0.0
appendicitis 50%	0.0003	0.0042	0.2
auto 20%	0.0001	0.0089	0.4286
backache 20%	0.0	0.0	0.0
backache 50%	1.0	0.0004	0.2308
biomed 20%	0.0004	0.0	0.2857
biomed 50%	0.0148	0.2777	0.625
breast-cancer 20%	0.0138	0.0003	0.1667
cars 20%	0.0079	0.0033	0.25
cleveland-nominal 20%	0.0001	0.0	0.0
cleveland-nominal 50%	0.6296	0.0	0.1667
cleveland 20%	0.094	0.0252	0.4167
cleve 20%	0.299	0.0005	0.2727
cloud 20%	0.0001	0.0	0.0
cloud 50%	0.8602	0.1111	0.1111
dermatology 20%	0.0001	0.0	0.0
dermatology 50%	0.0002	0.0	0.0
ecoli 20%	0.0001	0.0	0.3333
ecoli 50%	0.0062	0.143	0.4
glass2 20%	0.0005	0.0016	0.2857
glass 20%	0.001	0.0032	0.375
haberman 20%	0.3309	0.0013	0.3158
haberman 50%	1.0	0.0382	0.3
hayes-roth 20%	0.0	0.0	0.0
hayes-roth 50%	0.0006	0.0	0.25
heart-c 20%	0.0412	0.0016	0.2222
heart-statlog 20%	0.0646	0.0063	0.2222
hepatitis 20%	0.0001	0.0	0.2
hepatitis 50%	0.4735	0.1289	0.4167
hungarian 20%	0.8897	0.0361	0.4286
lupus 20%	0.0	0.0	0.0
lupus 50%	0.0002	0.0001	0.2
lymphography 20%	0.0001	0.0	0.2857
lymphography 50%	0.1919	0.0011	0.2308
molecular_biology_promoters 20%	0.0	0.0	0.0
molecular_biology_promoters 50%	0.0046	0.0022	0.2
new-thyroid 20%	0.0	0.0	0.0
new-thyroid 50%	0.0005	0.0029	0.1667
postoperative-patient-data 20%	0.0	0.0	0.0
postoperative-patient-data 50%	0.0004	0.0	0.3571
soybean 20%	0.0008	0.0	0.1429
soybean 50%	0.1325	0.0029	0.1
spect 20%	0.0001	0.0	0.125
spect 50%	0.488	0.0005	0.1765
tae 20%	0.0	0.0	0.1667
tae 50%	1.0	0.0016	0.3478

Table 6: Confined Primal Integral (CPI) values for all 46 benchmark instances whose optimal tree size is known. Values are normalized to a [0, 1] range (lower is better): A value of 0 indicates the method produced the optimal tree immediately. Values are rounded to 4 decimals. ($\iota = 0.10$)

D Survival Analysis

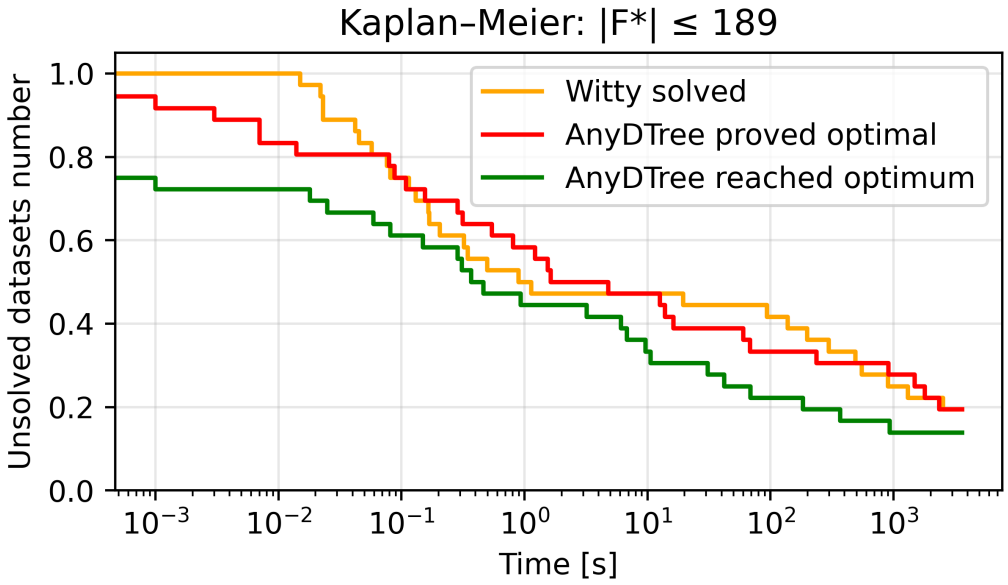

Figure 3: Kaplan-Meier curve for datasets with $|F^*| \leq 189$. Lower is better.

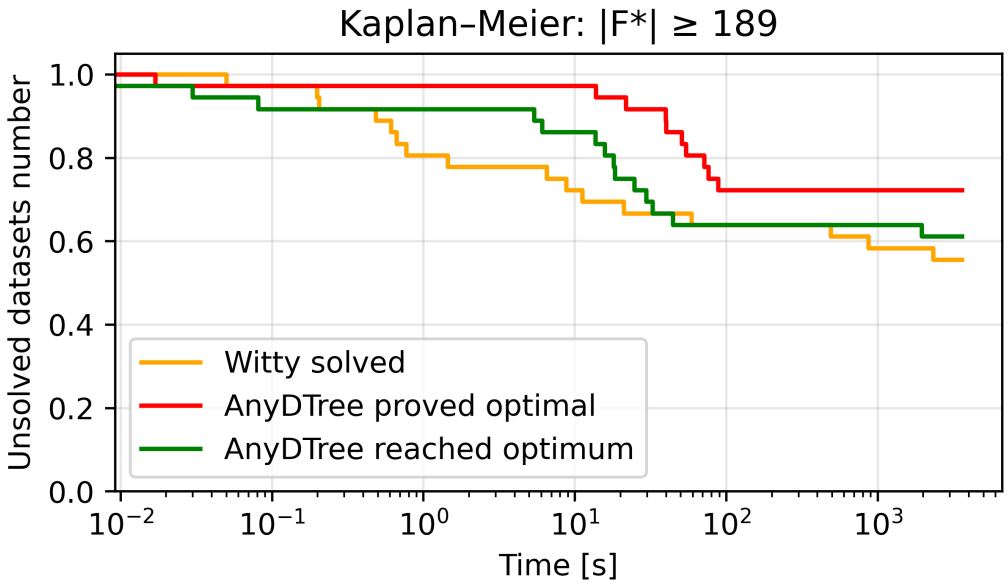

Figure 4: Kaplan-Meier curve for datasets with $|F^*| \geq 189$. Lower is better.

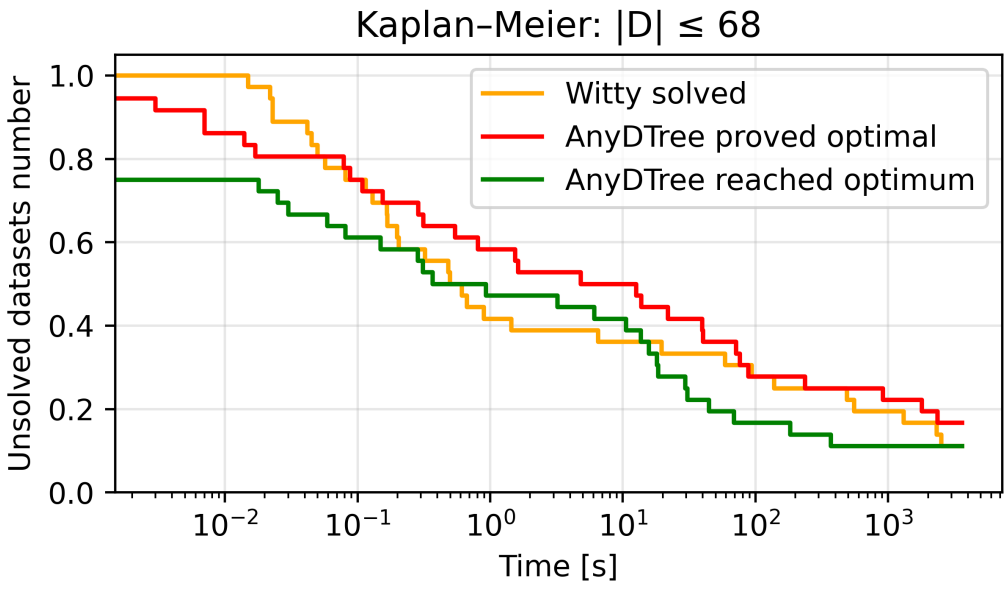

Figure 5: Kaplan-Meier curve for datasets with $|D| \leq 68$. Lower is better.

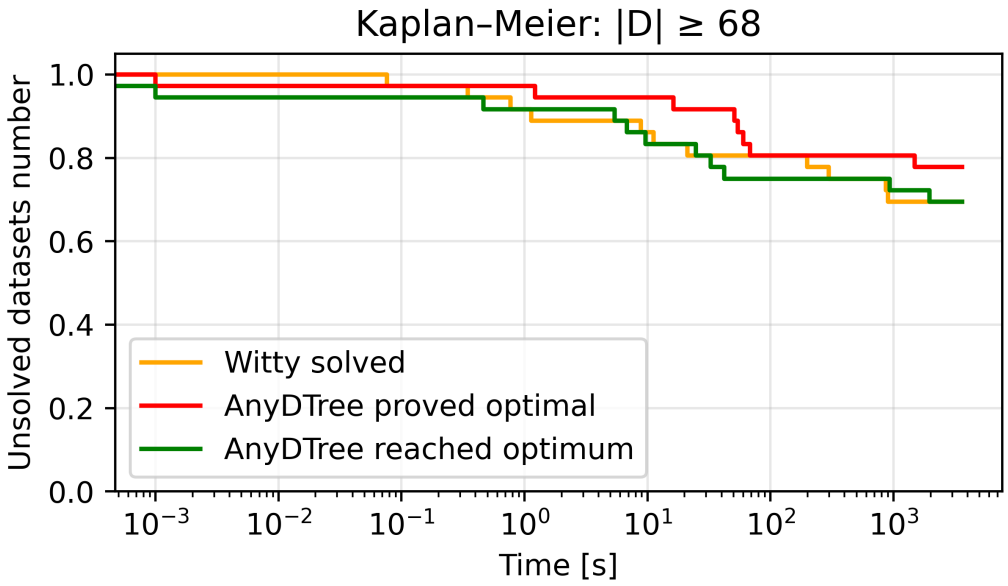

Figure 6: Kaplan-Meier curve for datasets with $|D| \geq 68$. Lower is better.

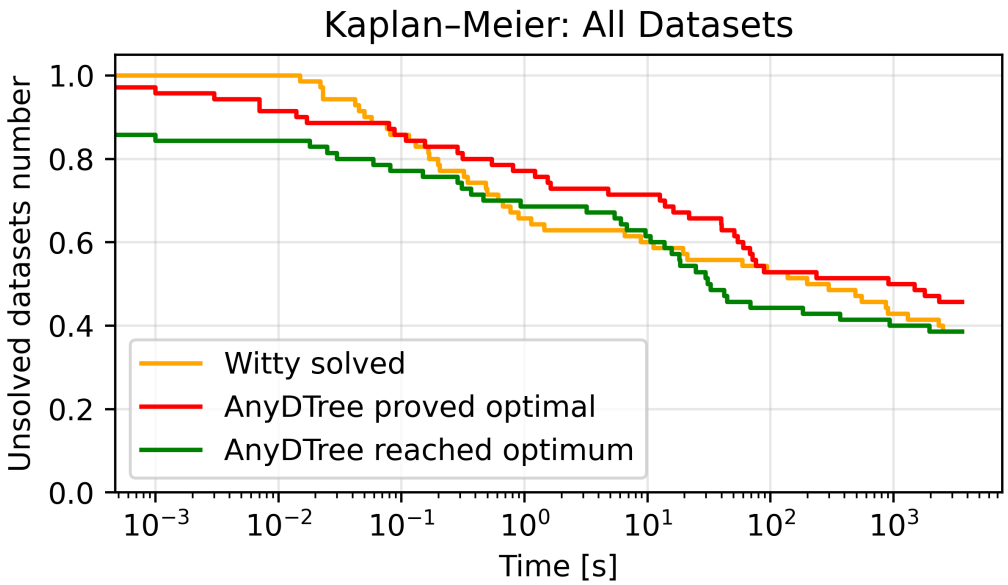

Figure 7: Kaplan-Meier curve for the full benchmark (all 70 datasets). Lower is better.

References

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

[2] Tobias Berthold and Zsolt Csizmadia. The confined primal integral: A measure to benchmark heuristic MINLP solvers against global MINLP solvers. *Mathematical Programming*, 188:523–537, 2021.

[3] Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106(7):1039–1082, 2017.

[4] L. Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth & Brooks, Monterey, CA, 1984.

[5] Catalin E. Brita, Jacobus G. M. van der Linden, and Emir Demirović. Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence*, 2025.

[6] Emir Demirović, Emmanuel Hebrard, and Louis Jean. Blossom: An anytime algorithm for computing optimal decision trees. In *Proceedings of the 40th International Conference on Machine Learning*, pages 7533–7562, 2023.

[7] Emir Demirović, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J. Stuckey. MurTree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

[8] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[9] Christian Komusiewicz, Pascal Kunz, Frank Sommer, and Manuel Sorge. On computing optimal tree ensembles. In *Proceedings of the 40th International Conference on Machine Learning*, pages 17364–17374, 2023.

[10] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and João Marques-Silva. Learning optimal decision trees with SAT. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 1362–1368, 2018.

[11] Siegfried Nijssen and Elisa Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 530–539, 2007.

[12] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[13] Luca Pascal Staus, Christian Komusiewicz, Frank Sommer, and Manuel Sorge. Witty: An efficient solver for computing minimum-size decision trees. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence*, 2025.

[14] Jacobus G. M. van der Linden, Mathijs M. de Weerdt, and Emir Demirović. Necessary and sufficient conditions for optimal decision trees using dynamic programming. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, 2023.

[15] Jacobus G. M. van der Linden, Daniël Vos, Mathijs M. de Weerdt, Sicco Verwer, and Emir Demirović. Optimal or greedy decision trees? Revisiting their objectives, tuning, and performance, 2025.

[16] Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence*, pages 1625–1632, 2019.