The Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI-25)

Witty: An Efficient Solver for Computing Minimum-Size Decision Trees

Luca Pascal Staus[1], Christian Komusiewicz[1], Frank Sommer[2], Manuel Sorge[2]

[1]Institute of Computer Science, Friedrich Schiller University Jena, Germany
[2]Institute of Logic and Computation, TU Wien, Austria
luca.staus@uni-jena.de, c.komusiewicz@uni-jena.de, fsommer@ac.tuwien.ac.at, manuel.sorge@ac.tuwien.ac.at

Abstract

Decision trees are a classic model for summarizing and classifying data. To enhance interpretability and generalization properties, it has been proposed to favor small decision trees. Accordingly, in the minimum-size decision tree training problem (MSDT), the input is a set of training examples in $\mathbb{R}^d$ with class labels and we aim to find a decision tree that classifies all training examples correctly and has a minimum number of nodes. MSDT is NP-hard and therefore presumably not solvable in polynomial time. Nevertheless, a promising algorithmic paradigm called *witness trees* which solves MSDT efficiently if the solution tree is small has been developed. In this work, we test this paradigm empirically. We provide an implementation, augment it with extensive heuristic improvements, and scrutinize it on standard benchmark instances. The augmentations achieve a mean 324-fold (median 84-fold) speedup over the naive implementation. Compared to the state of the art they achieve a mean 32-fold (median 7-fold) speedup over the dynamic programming based MurTree solver and a mean 61-fold (median 25-fold) speedup over SAT-based implementations. As a theoretical result we obtain an improved worst-case running-time bound for MSDT.

Full version on arXiv — http://arxiv.org/abs/2412.11954
Code, Data and Experimental Results —
https://doi.org/10.5281/zenodo.11235017

1 Introduction

Traditional decision trees recursively partition the feature space with axis-parallel binary cuts and assign a class to each part of the partition. When learning decision trees, we are given a training data set that consists of examples $E \subseteq \mathbb{R}^d$ labeled by classes and we want to find a decision tree that classifies the training data set (see Section 2 for the formal definitions). In addition, we want to optimize certain criteria, chiefly among them we want to minimize the number of cuts in the tree (or, equivalently, the size of the tree).[1] This is because small trees are more easily interpretable (Molnar 2020; Moshkovitz et al. 2020) and they are thought to be more accurate on unknown data (Fayyad and Irani 1990; Hu et al. 2020). The resulting optimization problems are NP-hard, however (Hyafil and Rivest 1976; Ordyniak and Szeider 2021). The use of heuristics such as CART (Breiman et al. 1984) is therefore widespread. Recent advancements in hardware and algorithms have made it feasible to compute provably optimal trees for training data sets with up to several hundreds of examples (see the surveys by Carrizosa, Molero-Río, and Romero Morales (2021); Costa and Pedreira (2023) and a full list of 26 implementations in the full version). The implementations still suffer from poor scalability, however (Costa and Pedreira 2023). In this paper we contribute towards overcoming this limitation.

To that end, as has been done before (Janota and Morgado 2020; Narodytska et al. 2018; Demirovic et al. 2022), we focus on the training problem in which we want to find a minimum-size perfect decision tree for the input training examples from two classes. *Perfect* means that all examples are classified correctly. This choice is to create a baseline method and we aim to later extend our approach to weaker accuracy guarantees and more classes. We build on recent theoretical algorithmic research that investigated what properties of the training data make computing smallest decision trees hard or tractable (Ordyniak and Szeider 2021; Eiben et al. 2023; Kobourov et al. 2023; Komusiewicz et al. 2023).

Our starting point is the algorithmic paradigm of *witness trees*, introduced by Komusiewicz et al. (2023). This paradigm gives an algorithm with the current-best running time bound of $\mathcal{O}((6\delta D s)^s \cdot sn)$ where n is the number of training examples, D is the largest domain size, $\delta \leq d$ is the largest number of dimensions in which two training examples with different classes differ, and s is the number of cuts in the solution tree. This bound guarantees that the algorithm is fast whenever D, δ, and s are reasonably small. For D this is very often the case, in fact many data sets have only binary dimensions. The cut number s should be small since we aim for interpretable models. Finally, δ was shown to be reasonably small in benchmark data (Ordyniak and Szeider 2021), see also Table 3 in the full version. Thus, the running time bound hints at practical usefulness of the paradigm. Equally important for us is that the paradigm simplified previous theoretical approaches (Ordyniak and Szeider 2021; Eiben et al. 2023) to such an extent as to make it promising as a foundation of a competitive solver. Addition-

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]Another popular choice is to minimize the depth of the tree. We opted for the size criterion instead for simplicity. Our techniques are broadly applicable to the depth criterion, see the discussion in Section 8.

ally, the approach generalizes to other learning models and even ensembles (Komusiewicz et al. 2023; Ordyniak et al. 2024), making it particularly interesting to test in practice.

The main idea of the witness-tree paradigm is to start with a trivial decision tree consisting of a single leaf labeled by an arbitrary class. Then we pick a training example which is classified incorrectly, we call it *dirty*, and we recursively try in a branching over all possibilities to refine the current tree with a new cut and a new leaf in which the dirty example is classified correctly. We label the new leaf with the dirty example as *witness* and mandate that all future refinements maintain all witnesses at their assigned leafs. This reduces the search space that we need to explore to find the desired decision tree; see Section 3 for a more detailed description. In this work, we provide an implementation of the above paradigm and we extensively augment it with improvements that give heuristic speedups while maintaining the optimality of the computed decision tree.

Our main result is that these improvements substantially speed up the algorithm to the point where the resulting solver, called `Witty`, performs substantially better than the state of the art (see Section 7). To the best of our knowledge, the state-of-the-art solvers for computing minimum-size perfect decision trees are SAT-based solvers by Narodytska et al. (2018) and Janota and Morgado (2020) and the dynamic-programming algorithm `MurTree` by Demirovic et al. (2022). All other implementations for computing variants of optimal decision trees solve different optimization problems and are not suitable for the task discussed in the present paper, see the full version for details.

We tested the solvers on standard benchmark data sets from the Penn Machine Learning Benchmarks (Romano et al. 2022). Out of the three state-of-the-art solvers, `MurTree` performed the best, solving 371 out of the 700 instances. In comparison, `Witty` was able to solve 388 instances and achieves a mean 32-fold (median 7-fold) speedup[2] over `MurTree`. Compared to `MurTree` `Witty` performs especially well on instances that have dimensions with a large number of different values. This is due to the fact that `Witty` does not require binary dimensions and one of our above-mentioned improvements specifically exploits large domains.

To achieve these results, we employ the following techniques. First, we apply data reduction rules that simplify the instances (Section 3). Second, we improve the branching rule which introduces the next cut into the partial decision tree in several ways (Section 3): We carefully select which dirty examples to use for branching, specify a suitable sequence of the cuts to try, and observe that some cuts can be omitted from branching. Third, we introduce lower-bounding strategies that determine a minimum number of cuts that still need to be added to the tree, to further shrink the search space (Section 4). Fourth, we use symmetry-breaking techniques which we call subset constraints that leverage information from unsuccessfully returning branches which essentially fixes some examples into subtrees of the solution decision tree (Section 5). This also yields an improved running-time bound of $\mathcal{O}((\delta D \log s)^s \cdot sn)$ (Theorem 5.9). Finally, during the search we select certain subsets of examples for which we directly compute lower bounds, cache them, and exploit them later in the search (Section 6). We rigorously analyze all of the above techniques and prove them to preserve optimality of the computed trees.

Apart from improved scalability, other key advantages of our implementation include that, different from the state-of-the-art algorithms, it is not necessary to binarize the dimensions, and that the general paradigm is very flexible so that it can be adapted for reoptimizing a given decision tree, for further optimization goals such as minimizing the number of misclassified examples, and for other models related to decision trees such as decision lists. Similarly, the above heuristic improvements are flexible and they apply to computing minimum-depth trees, for example. In summary, we provide an implementation and extensive heuristic tuning of the witness-tree paradigm for computing minimum-size decision trees which is flexible and at the same time substantially outperforms the state of the art.

2 Preliminaries

For $n \in \mathbb{N}$, we denote $[n] := \{1, 2, \dots, n\}$. For a vector $x \in \mathbb{R}^d$, we denote by $x[i]$ the ith entry in x. Let Σ be a set of *class symbols*. We consider binary classification, so we assume that $\Sigma = \{\text{blue}, \text{red}\}$. A *data set* with classes Σ is a tuple (E, λ) of a set of *examples* $E \subseteq \mathbb{R}^d$ and their class labeling $\lambda\colon E \to \Sigma$. Note that this formulation captures ordered dimensions because such dimensions can be mapped into $\mathbb{R}$. We assume that (E, λ) does *not* contain two examples with identical coordinates but different class labels. For a fixed data set (E, λ), we let $n := |E|$ denote the *number of examples* and d the *dimension* of the data set.

For each dimension $i \in [d]$, we let $\mathsf{Thr}(i)$ be a smallest set of *thresholds* such that for each pair of examples $e_1, e_2 \in E$ with $e_1[i] < e_2[i]$ there is a threshold $t \in \mathsf{Thr}(i)$ with $e_1[i] \leq t < e_2[i]$. Note that such a set $\mathsf{Thr}(i)$ can be computed in $\mathcal{O}(n \log n)$ time and that $|\mathsf{Thr}(i)| \leq D$. A *cut* is a pair (i, t) where $i \in [d]$ is a dimension and $t \in \mathsf{Thr}(i)$ is a threshold in dimension d. The set of all cuts is denoted by $\mathsf{Cuts}(E)$. The *left side* of a cut with respect to $E' \subseteq E$ is $E'[\leq (i,t)] := \{e \in E' \mid e[i] \leq t\}$, and the *right side* of a cut with respect to E' is $E'[> (i,t)] := \{e \in E' \mid e[i] > t\}$.

A *decision tree* is a tuple $\mathcal{D} = (T, \mathsf{cut}, \mathsf{cla})$ where T is an ordered rooted binary tree with vertex set $V(\mathcal{D})$, that is, each inner vertex has a well-defined left and right child. Furthermore, $\mathsf{cut} : V(\mathcal{D}) \to \mathsf{Cuts}(E)$ maps every inner vertex to a cut and $\mathsf{cla} : V(\mathcal{D}) \to \Sigma$ labels each leaf. The *size* of $\mathcal{D}$ is the number of inner vertices. For each vertex $v \in V(\mathcal{D})$ we define a set $E[\mathcal{D}, v] \subseteq E$ of examples that are *assigned* to v. If v is the root of $\mathcal{D}$, then $E[\mathcal{D}, v] := E$. For other vertices, the assigned values are defined via the cuts at inner vertices. More precisely, for a parent vertex p with left child u and right child v we set $E[\mathcal{D}, u] := E[\mathcal{D}, p][\leq \mathsf{cut}(p)]$ and $E[\mathcal{D}, v] := E[\mathcal{D}, p][> \mathsf{cut}(p)]$. If $\mathcal{D}$ is clear from

[2]For the calculation of all speedup factors in this paper we ignored all instances that were solved in less than one second by both algorithms as these instances are not a good indicator for the scaling behavior of the running times.

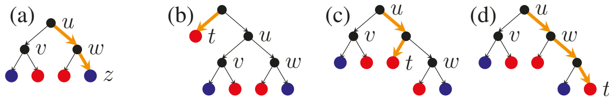

Figure 1: Examples of one-step refinements. The orange path is the classification path of a red example e. (a) A witness tree W where e is misclassified in leaf z. (b), (c), (d) Three possible trees resulting from one-step refinements of W where e is the witness of the new leaf t.

the context, we just write $E[v]$. By definition, each example $e \in E$ is *assigned* to exactly one leaf ℓ of $\mathcal{D}$. We say that ℓ is *the leaf of* e in $\mathcal{D}$ and denote ℓ by $\mathsf{leaf}(\mathcal{D}, e)$ or just $\mathsf{leaf}(e)$ if $\mathcal{D}$ is clear. An example $e \in E$ is *dirty* in T if we have $\lambda(e) \neq \mathsf{cla}(\ell)$ with ℓ being the leaf of e. The set of all dirty examples in T is $\mathsf{Dirty}(T)$. A decision tree *classifies* (E, λ) if the class of every example $e \in E$ matches the class of its leaf, that is, we have $\lambda(e) = \mathsf{cla}(\mathsf{leaf}(e))$ for all $e \in E$. In this case we call $\mathcal{D}$ *perfect*.

We aim to solve the following computational problem.

MINIMUM-SIZE DECISION TREE (MSDT)
Instance: A data set (E, λ).
Task: Find a smallest decision tree that classifies (E, λ).

We let BOUNDED-SIZE DECISION TREE (BSDT) denote the variant where we are also given a number s and need to find a decision tree of size at most s or decide that no such tree exists. We can solve MSDT by solving multiple instances of BSDT as follows: Start with $s = 1$. Increase s one by one and solve BSDT for each s until a tree is found. This tree must be a solution for MSDT.

Proofs marked with (★) are deferred to the full version.

3 Base Version – the Witness-Tree Algorithm

The base version of our algorithm is the witness-tree algorithm (Komusiewicz et al. 2023). In the following, we briefly explain the algorithm. The presented algorithm solves the BSDT problem where we have a fixed size threshold s. We then use this algorithm to solve MSDT as described above.

Witness trees and one-step refinements. Witness trees are decision trees augmented with a labeling of the leafs by examples. Formally, a *witness tree* is a tuple $W = (T, \mathsf{cut}, \mathsf{cla}, \mathsf{wit})$ where $(T, \mathsf{cut}, \mathsf{cla})$ is a decision tree and $\mathsf{wit} : V(W) \to E$ is a map that maps each leaf ℓ to an example $e \in E[\ell]$ of the same class as ℓ. These examples are called *witnesses* of W. During branching, W is extended by adding a new inner vertex and a new leaf. This is done via one-step refinements. Formally, a *one-step refinement* of W is a tuple (v, i, t, e) where $v \in V(W)$ is some vertex in W, (i, t) is a cut in $\mathsf{Cuts}(E)$, and $e \in E[W, v]$ is an example.

A one-step refinement (v, i, t, e) is applied to a witness tree W as follows (see Figure 1): First, subdivide the edge from v to its parent p by adding a new inner vertex u with cut (i, t). Now, p is the parent of u and u is the parent of v. If v was the left (right) child of p then u becomes the left (right) child of p. If v is the root of the tree (hence v has no parent p) then u becomes the new root. Second, add a new leaf ℓ as the second child of u (with v being the first child). The example e determines the assignment of ℓ and v to the sides of the cut (i, t): The example e needs to be assigned to leaf ℓ, so if e is assigned to the left child, then ℓ becomes the left child. Otherwise, ℓ becomes the right child. Finally, the class of ℓ is set to $\lambda(e)$ and the witness of ℓ is set to e.

Algorithm 1: Base witness-tree algorithm.

Input: A witness tree W, a data set (E, λ), and a maximum size $s \in \mathbb{N}$.
Output: A perfect witness tree of size at most s or $\perp$ if none could be found.

```
1 Function Refine (W, (E, λ), s)
2     if W classifies (E, λ) then return W
3     if W has size s then return ⊥
4     e ← some dirty example from Dirty(W)
5     forall r = (v, i, t, e) ∈ Ref(W) do
6         Apply r to W to create a new witness tree R
7         R' ← Refine (R, (E, λ), s)
8         if R' ≠ ⊥ then return R'
9     return ⊥
```

Let R denote the newly created witness tree. We write $W \xrightarrow{r} R$ to indicate that R was created by applying r to W. An application of a sequence $r_1, \ldots, r_q$ of one-step refinements to create R from W is denoted by $W \xrightarrow{r_1,\ldots,r_n} R$. $\mathsf{Ref}(W)$ is the set of all one-step refinements $r = (v, i, t, e)$ of W such that $e \in \mathsf{Dirty}(W)$ is dirty in W, e and $\mathsf{wit}(\mathsf{leaf}(e))$ are on different sides of the cut (i, t) and r does *not* change the leaf of any witness of W. This set is important because it contains all one-step refinements that are relevant for the algorithm.

Description of the algorithm solving BSDT. Algorithm 1 shows the pseudocode. For the correctness proof, refer to the work of Komusiewicz et al. (2023). Algorithm 1 starts with a witness tree W that consists of just one leaf ℓ. An arbitrary witness for this leaf ℓ is chosen and the class of ℓ is set to the class of this witness. Next, `Refine` is called (Line 1). Algorithm 1 then recursively chooses a dirty example e and iterates over all one-step refinements in $\mathsf{Ref}(W)$ in which e is the dirty example. The idea is that since e is currently dirty, we need to assign e to a new leaf with class $\lambda(e)$ since all examples are required to be correctly classified. The one-step refinements with e as dirty example do this by making e the witness of the newly added leaf and assigning the class of e to that leaf. Algorithm 1 traverses a search tree where each node represents a call of `Refine`. To avoid confusion, from now on we call the vertices of the search tree *nodes*, and the vertices in a decision/witness tree *vertices*. For a node N, we let $\mathsf{Tree}(N)$ denote the witness tree W that `Refine` is called with and we use $\mathsf{ex}(N)$ to refer to the dirty example that is chosen in Line 4.

We now describe some first improvements to this algorithm, leading to the first version of the solver.

Dirty example priority. For the correctness it is not relevant which dirty example is chosen in Line 4. This permits our first improvement: choose a dirty example that minimizes the number of branches. Ideally one can choose the dirty example such that the number of one-step refinements is as small as possible. Computing this for every dirty example for every call of `Refine` takes too long, however. Hence, we update this number for an example e only whenever e is assigned to a new leaf; preliminary experiments showed that this is a good trade-off. We can use the same idea to choose the witness of the initial leaf and the initial dirty example. That means before the algorithm starts we can find the pair of examples with different classes that minimizes the number of one-step refinements, that is, the number of cuts separating them and choose one of these two elements as initial witness.

Data reduction. In the following, we briefly describe the used rules. Correctness proofs and their experimental evaluation can be found in the full version. First, if there are two examples e_1 and e_2 having the same value in each dimension (recall that e_1 and e_2 are required to have the same class label), we remove one of them from (E, λ). Second, if there is a dimension i such that all examples have the same value in this dimension, we remove i. Third, if the instance has two equivalent cuts, then remove one of them. Here, two cuts $(i_1, t_1), (i_2, t_2) \in \mathsf{Cuts}(E)$ are *equivalent* if $E[\leq (i_1, t_1)] = E[\leq (i_2, t_2)]$. For the fourth rule, consider a pair of cuts (i, t_1) and (i, t_2) in the same dimension i with $t_1 < t_2$. Now, if all examples on the left (right) side of both cuts have the same class, then remove (i, t_1) (or (i, t_2) respectively). For the last rule, consider a pair of dimensions i_1 and i_2. If there is an ordering of the examples such that the values of the examples do not decrease in both dimensions, then one can construct a new dimension $i_{1,2}$ such that for each cut in dimensions i_1 or i_2, there is an equivalent cut in dimension $i_{1,2}$ and vice versa. Now, i_1 and i_2 are replaced by $i_{1,2}$.

4 Lower Bounds

We introduce two lower bounds that we use to prune the search tree. In both lower bounds an instance of SET COVER is constructed and then a lower bound is calculated for this instance. In SET COVER the input is a universe U and a family S of subsets of U, and the task is to compute the smallest integer k such that there is a subset $S' \subseteq S$ of size exactly k whose union is the universe U.

We show that for our two lower bounds, the size k of the smallest subset S' is a lower bound for the minimum number of one-step refinements that are needed to correctly classify all examples in the current witness tree. We calculate these lower bounds after Line 5 in each call of `Refine` in Algorithm 1. If k is bigger than s minus the current size of W we can return $\perp$. Since SET COVER is NP-hard (Karp 1972), calculating the exact value of k in every call of `Refine` is not feasible. Instead we are going to introduce different ways of calculating a lower bound for k.

First lower bound: Improvement Lower Bound (ImpLB). The idea of the ImpLB is to find the minimum number of one-step refinements that are necessary to correctly classify only the examples that are dirty in the current witness tree W while ignoring the examples that are already correctly classified. Moreover, we ignore that one-step refinements may interfere with each other and consider the effect of each one-step refinement separately. To assess this effect, we use the following definition.

Definition 4.1. Let W be a witness tree, $E' \subseteq \mathsf{Dirty}(W)$ a set of dirty examples, and $r \in \mathsf{Ref}(W)$ a one-step refinement with $W \xrightarrow{r} R$. Then we define

$$\mathsf{imp}(W, E', r) \coloneqq \{e' \in E' \mid e' \notin \mathsf{Dirty}(R)\}$$

as the set of dirty examples that get correctly classified by r. We call these sets the *imp sets*.

We now create an instance of SET COVER by choosing $\mathsf{Dirty}(W)$ as the universe and the family of subsets $\{\mathsf{imp}(W, \mathsf{Dirty}(W), r) \mid r \in \mathsf{Ref}(W)\}$. To show that the solution of this instance is a lower bound, we show that for any series of s one-step refinements I' that lead to W being perfect there is a set $I \subseteq I'$ of size at most s such that the union of all sets in I is equal to $\mathsf{Dirty}(W)$.

Theorem 4.2 (★). *Let $W := R_0$ be a witness tree and $(r_1, \ldots, r_s)$ a series of one-step refinements with $R_{i-1} \xrightarrow{r_i} R_i$, $r_i \in \mathsf{Ref}(R_{i-1})$ for $i \in [s]$ such that R_s is perfect. Then there is a set $I \subseteq \mathsf{Ref}(W)$, $|I| \leq s$, such that*

$$\bigcup_{r \in I} \mathsf{imp}(W, \mathsf{Dirty}(W), r) = \mathsf{Dirty}(W).$$

Calculating the ImpLB. Instead of considering the content of each set, we only consider their sizes. Thus, we calculate the smallest set of subsets such that the sum of their sizes is bigger or equal to the size of the universe. Hence, to calculate the ImpLB we just need to look at each one-step refinement $r \in \mathsf{Ref}(W)$, calculate the size of $\mathsf{imp}(W, \mathsf{Dirty}(W), r)$, sort the sizes in descending order, and then check how many sets are needed to reach a sum that is at least the size of $\mathsf{Dirty}(W)$. We apply the ImpLB in each search-tree node.

Second lower bound: Pair Lower Bound (PairLB). For the PairLB we use a similar idea as for the ImpLB but instead of looking at sets of dirty examples we look at sets of pairs of examples that are assigned to the same leaf but have different classes. For each one-step refinement r we define a *pairsplit set* as the set of all pairs that are split up by r, that is, all pairs where the two examples are no longer in the same leaf of the tree created by r. With the set of all pairs as the universe and all pairsplit sets as the family of subsets we obtain a SET COVER instance. Similar to the ImpLB, the solution of this instance is a lower bound for the minimum number of one-step refinements that are needed to correctly classify all examples in the current witness tree.

However, preliminary experiments showed that calculating the PairLB in every search-tree node does not improve the running time of the algorithm. Thus, we use the PairLB only to calculate an initial lower bound for the solution of MSDT. For this we use the LP-relaxation of the standard ILP-formulation of SET COVER. Further details and proofs for the PairLB are in the full version.

5 Subset Constraints

A *subset constraint of a vertex* v in a witness tree W is a subset $S \subseteq E[W, v]$ of examples which imposes the constraint that one-step refinements are not allowed to remove all examples of S from the subtree of v. The idea is that if such one-step refinements lead to a perfect tree, then there is a different perfect tree that does not violate this constraint and is not larger. For example, if one can replace threshold t in vertex v of W by a different threshold t' such that there is no example in v with a threshold between t and t', we only have to test one of t and t'. Testing this for each pair of thresholds is too inefficient. Instead, we use subset constraints:

Definition 5.1. Let W be a witness tree and let $v \in V(W)$. We call a subset $C \subseteq E$ a *Subset Constraint* of v. We call C *violated* if $E[W, v] \cap C = \emptyset$. The set $\mathsf{Const}(W, v)$ contains all subset constraints of v in the tree W.

Subset constraints are added continuously during the algorithm. After each one-step refinement, we update for each node v which examples are still in the subtree of a node v, thus immediately detecting any violated subset constraint.

Correctness of subset constraints. We add different subset constraints after the application of one-step refinements. To show that the addition of these constraints is safe, we need the following definition.

Definition 5.2. Let $W = (T, \mathsf{cut}, \mathsf{cla}, \mathsf{wit})$ and $R = (T', \mathsf{cut}', \mathsf{cla}', \mathsf{wit}')$ be two witness trees. We say that R is a *refinement* of W if and only if W and R fulfill the following properties:

1. $V(W) \subseteq V(R)$.
2. A vertex $v \in V(W)$ is a leaf in W if and only if v is a leaf in R, and for each leaf $\ell \in V(W)$ we have $\mathsf{cla}(\ell) = \mathsf{cla}'(\ell)$ and $\mathsf{wit}(\ell) \in E[R, \ell]$.
3. Let $v_1, v_2 \in V(W)$ be a pair of vertices. Vertex v_1 is in the left subtree of v_2 in W if and only if v_1 is in the left subtree of v_2 in R. Vertex v_1 is in the right subtree of v_2 in W if and only if v_1 is in the right subtree of v_2 in R.
4. For every inner vertex $v \in V(W)$ we have $\mathsf{cut}(v) = \mathsf{cut}'(v)$.

If R was created by applying one or more one-step refinements to W then R is a refinement of W. In the full version, we show that the reverse direction is also true.

We can now define what it means for a subset constraint to be safe. Note that this definition depends on a fixed order in which the algorithm considers one-step refinements, the description of this order is deferred to the full version.

Definition 5.3. Let N be a node in the search tree with witness tree W and the dirty example $e := \mathsf{ex}(N)$ and let $r := (v, i, t, e) \in \mathsf{Ref}(W)$ be a one-step refinement with $W \xrightarrow{r} W'$ that introduces a subset constraint C for the newly added inner vertex.

We say that C is *correct* if for each perfect witness tree R that is a refinement of W' and violates C, there is a different witness tree R' with the following properties:

1. R' is perfect.
2. R' is not bigger than R.
3. There is a different one-step refinement $r' \in \mathsf{Ref}(W)$ with $W \xrightarrow{r'} W''$ that Algorithm 1 chooses before r such that R' is a refinement of W''.

Theorem 5.4 (★). *If there is a perfect witness tree of size at most $s \in \mathbb{N}$, then there is a perfect witness tree of size at most s that does not violate any correct subset constraints.*

Next, we introduce two specific subset constraints.

Threshold Subset Constraints. Our first subset constraint is based on the observation that replacing a threshold by another threshold without changing the leaf assignment of any example can be possible.

Definition 5.5. Let N be a node in the search tree with the witness tree $W := \mathsf{Tree}(N)$ and the dirty example $e := \mathsf{ex}(N)$, let $r = (v, i, t, e), r' = (v, i, t', e) \in \mathsf{Ref}(W)$ with $W \xrightarrow{r} R$ and $W \xrightarrow{r'} R'$ such that we have $e[i] \leq t' < t$ or $t < t' < e[i]$, and let ℓ and u be the leaf and inner vertex that are added to W by r. We add the *Threshold Subset Constraint* $C := E[R, \ell] \setminus E[R', \ell]$ to $\mathsf{Const}(R, u)$ and call t' the *constraint threshold* of C.

If a Threshold Subset Constraint of a vertex u in W is violated then replacing the threshold of u by the constraint threshold t' does not change the leaf of any example in W.

Theorem 5.6 (★). *Threshold Subset Constraints are correct.*

Dirty Subset Constraints. Our second subset constraint is based on the idea that we discard some vertices on the leaf-to-root path for one-step refinements. For example, we may discard one-step refinements r at a vertex v when one of the child subtrees of v is already perfect, since it is also possible to apply r at the root of the other child subtree.

Definition 5.7. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \mathsf{Tree}(N)$ and the dirty example $e := \mathsf{ex}(N)$, let v be an inner vertex in W with the children v_1 and v_2, let $r = (v, i, t, e), r' = (v_1, i, t, e) \in \mathsf{Ref}(W)$ be two one-step refinements with $W \xrightarrow{r} R$ and $W \xrightarrow{r'} R_1$, let ℓ and u be the leaf and inner vertex that are added to W by r. We add the *Dirty Subset Constraint* $C = E[W, v_2] \cap \mathsf{Dirty}(W)$ to $\mathsf{Const}(R, u)$.

Theorem 5.8 (★). *Dirty Subset Constraints are correct.*

Now, we show that, using *Dirty Subset Constraint*s, we can improve on the running time of $\mathcal{O}((\delta \cdot D \cdot s)^s \cdot s \cdot n)$ for MINIMUM-SIZE DECISION TREE shown by Komusiewicz et al. (2023) by replacing the s in the base of the running time by $\log(s)$.

Theorem 5.9 (★). MINIMUM-SIZE DECISION TREE *can be solved in $\mathcal{O}((\delta \cdot D \cdot \log(s))^s \cdot s \cdot n)$ time.*

6 Subset Caching

Our final improvement is inspired by the caching of subproblems used in MurTree (Demirovic et al. 2022) to solve the related problem of minimizing the number of misclassifications under a size and depth constraint. They iterate

over all possible cuts for the root and then calculate optimal solutions for the left and right subtree recursively. For this approach, caching of subproblems is very natural.

Since `Witty` can modify any part of the current witness tree, `Witty` is not directly amenable to caching. Thus, we modify the caching of subproblems as follows: We use a *set-trie data structure* (Savnik 2013) to store lower bounds for specific subsets of the examples. These lower bounds tell us how many inner vertices we need at least to correctly classify that subset of the examples. In any search-tree node N with $W := \mathsf{Tree}(N)$, we then look at each leaf $\ell \in V(W)$ and the set of examples $L := E[W, \ell]$ assigned to ℓ and check if a subset S of L is present in our data structure. If this is the case, then the lower bound z associated with S is also a lower bound for L. Since all examples in L are assigned to the same leaf, this is also a valid lower bound for W. If z is bigger than the remaining size budget s', we know that it is not possible to correctly classify the data set by applying at most s' one-step refinements to W. Consequently, the algorithm can return $\perp$. Otherwise, if z is not big enough, we keep checking our data structure for another subset of L until we either find a sufficiently large lower bound or until no more subsets of L are left.

Since our algorithm does not naturally calculate lower bounds for subsets of examples, to populate the set trie $\mathcal{T}$ we do the following: In a search-tree node N, we look at the example set $L := E[\mathsf{Tree}(N), \ell]$ of any leaf ℓ with $|L| \leq \min(|E|/4, 30)$. Let s' be the remaining size budget of the current witness tree. We run a separate instance of our algorithm that checks whether the set L can be correctly classified with a tree of size at most s'. If not, then we can add L to $\mathcal{T}$ with the lower bound $s' + 1$. Of course we could then check if L can be correctly classified with a tree of size at most $s' + 1$ to improve the lower bound, but we decided to only check size s' because that is sufficient to show that the current tree cannot correctly classify the data. If we ever need a better lower bound for L at a later point, we can still calculate it then. A limit of the size of L is necessary since otherwise one solves the original problem; preliminary experiments showed that $\min(|E|/4, 30)$ is a good limit.

A set-trie is especially useful when solving MSDT since we can reuse the set-trie for each instance of BSDT until we find the optimal s. For all solved instances, each trie had at most 80 000 vertices, resulting in a maximum space consumption of 3 GB.

7 Experimental Evaluation

Experimental setup. For our experiments, we used 35 data sets that were also used in the experimental evaluation of the state-of-the-art SAT-based MSDT solver (Janota and Morgado 2020). The data sets are part of the Penn Machine Learning Benchmarks (Romano et al. 2022). The full version contains an overview.

As Janota and Morgado (2020), we transformed the data sets as follows to meet the requirements of MSDT inputs: First, to ensure that all dimensions are ordered, we replaced each categorical dimension by a set of new binary dimensions indicating whether an example is in the category. Second, we converted each instance into a binary classification problem. For this, we colored all examples of the largest class red, and all remaining examples blue. Finally, if two examples had the same value in all dimensions but are colored differently, we removed one of them arbitrarily.

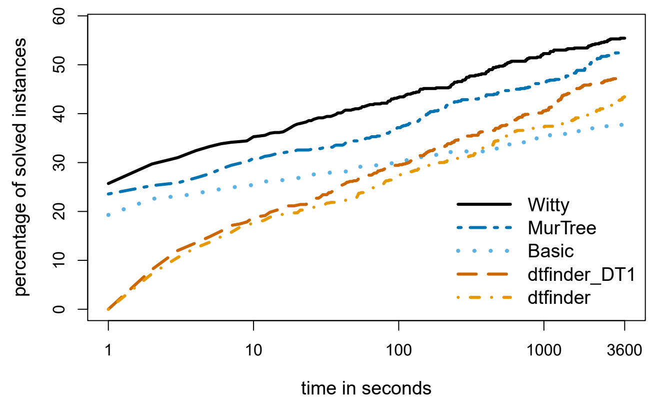

Figure 2: Comparison of different algorithms for MSDT. For each time t it is shown how many instances were solved by each algorithm in less than t seconds.

Following Janota and Morgado (2020), we randomly sampled multiple subsets of the examples from each data set. Specifically, for each data set we chose 10 random subsets with 20% of the examples and 10 random subsets with 50% of the examples. In total we ran our experiments on 700 instances with a time limit of 60 minutes for each instance.

Our experiments were performed on servers with 24 GB RAM and two Intel(R) Xeon(R) E5540 CPUs with 2.53 GHz, 4 cores, 8 threads, running Java openjdk 11.0.19. Each individual experiment was allowed to use up to 12 GB RAM. All algorithms were executed on a single core. However, we reserved two additional cores for each experiment to avoid side effects of other threads like the Java garbage collector. We implemented our algorithm in Kotlin (see Zenodo). To solve the LP-relaxation of the PairLB we used Gurobi 10.0.3 (Gurobi Optimization, LLC 2023). Recall that to compute the speedup, we ignored instances that were solved in less than one second by both algorithms.

Comparison with the basic version of `Witty`. Figure 2 shows the performance of `Witty` and `Basic`. `Witty` is the final version of our algorithm using all improvements while `Basic` is just Algorithm 1 together with the dirty example priority and the data reduction rules from Section 3.

`Basic` solved 264 out of the 700 instances. In comparison, the naive version of our algorithm without any improvements only solved 222 instances and `Basic` is 26 times (median 9 times) faster on instances solved by both. This shows that the dirty example priority and data reduction rules from Section 3 already give a large speedup.

`Witty` solved 388 instances and is roughly 65 times (median 23 times) faster than `Basic` on instances that were solved by both, showing that the combination of all of our improvements yields a vast speedup.

A more detailed comparison of the different solver configurations can be found in the full version.

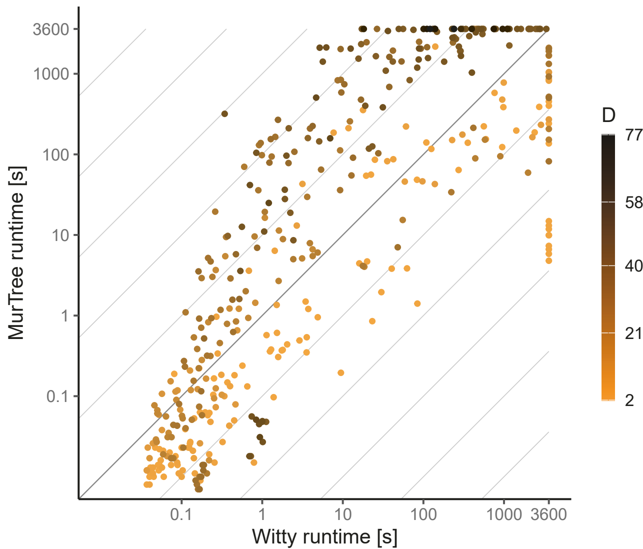

Figure 3: Comparison of the running times of Witty and MurTree for each instance with the color representing the largest domain size D.

Comparison with the state of the art. We compared Witty against the state-of-the-art SAT-based algorithms dtfinder_DT1 and dtfinder (Narodytska et al. 2018; Janota and Morgado 2020). We used the improved version of the encoding by Narodytska et al. (2018) that was presented by Janota and Morgado (2020). We also compared Witty against the state-of-the-art dynamic programming based solver, MurTree (Demirovic et al. 2022). Since these algorithms only support binary dimensions, we transformed the data sets following their approach by introducing a binary dimension for each cut indicating whether an example is on the left or right side of the cut. We still used the non-binarized data sets for Witty.

Figure 2 shows this comparison. Witty solved 57 instances more than dtfinder_DT1 which solved 26 instances more than dtfinder. Witty is roughly 61 times (median 25 times) faster than dtfinder_DT1. Furthermore, there is only one instance that can be solved by dtfinder or dtfinder_DT1 but not by Witty.

MurTree on the other hand solved 371 instances. However, only 341 of these instances were solved by both Witty and MurTree. On these instances, Witty achieved a mean 32-fold (median 7-fold) speedup over MurTree. One notable property of the instances that Witty solved faster than MurTree is that the largest domain size D tends to be a lot bigger than in the instances that MurTree could solve faster than Witty. This can be seen in Figure 3. We assume that the Threshold Subset Constraints from Section 5 are the reason for this: they are particularly effective on dimensions with many thresholds since they allow the algorithm to skip several thresholds within such a dimension.

We also looked at the values of the instance-specific parameters n, d, δ, c, and s of these instances. There are two notable observations. First, Witty also tends to perform better than MurTree on instances with a large number of cuts c. This is most likely related to the observation that these instances also tend to have a large value for D. Second, MurTree can solve instances with a larger optimal tree size s. MurTree solved instances with $s \leq 20$ while Witty could only solve instances with $s \leq 16$. This suggests that MurTree scales better with the size of the optimal tree as long as the largest domain size D is not too large. In fact, all instances with $s > 16$ that MurTree solved have a value of $D = 2$.

Comparison with heuristics. We evaluated the size, balance, and classification quality on test data of the trees computed by Witty against three heuristics: CART (Breiman et al. 1984), Weka (Frank, Hall, and Witten 2016), and YaDT (Ruggieri 2004, 2019). CART and Weka also compute perfect trees. Our evaluation (see the tables in the full version) shows that the trees of YaDT are much smaller than the trees of Witty which are a bit smaller than the ones of CART and Weka. Also, all computed trees are very balanced and achieve similar classification quality.

8 Outlook

We have provided a new fast solver Witty for computing perfect decision trees with a size constraint. While previous algorithms (Narodytska et al. 2018; Janota and Morgado 2020; Demirovic et al. 2022) only support binary dimensions, Witty in particular benefits from dimensions having many thresholds. We conclude with a set of limitations and possible extensions of Witty.

First, we focused on binary classification problems. However, Witty can easily be adapted for more classes: any example having a different class than that of the witness in any leaf is a dirty example; it would be interesting to scrutinize the resulting algorithm on multiclass instances.

Second, it is interesting to adapt and tune Witty to find decision trees with a depth constraint. While this constraint can easily be incorporated into Witty and all of our improvements are still valid, they could be made much tighter and many new improvements are possible that do not apply to the size constraint. For example, once a leaf with maximal depth has been found, the corresponding leaf-to-root path P cannot change anymore. The solution subtrees rooted at vertices in P thus have fixed example sets and can be determined independently of each other.

Finally, it is essential to adapt Witty to looser accuracy guarantees, for instance, to the scenario where a given number τ of misclassifications are allowed. Gahlawat and Zehavi (2024) showed that the underlying training problem is tractable in theory but the running time of their algorithm is impractical. The witness-tree paradigm could be extended to this problem by replacing dirty examples with a set of $\tau + 1$ dirty examples, of which one needs to be reclassified. While the theoretical running-time guarantee would increase, in practice this drawback could be outweighed by the fact that the size s of the optimal tree could decrease substantially (as shown by the results of YaDT on the benchmark data set).

Acknowledgments

This work is based on the first author's Master thesis (Staus 2024). Luca Pascal Staus was supported by the Carl Zeiss Foundation, Germany, within the project "Interactive Inference". Frank Sommer was supported by the Alexander von Humboldt Foundation and partially supported by the DFG, project EAGR (KO 3669/6-1).

References

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees*. Wadsworth.

Carrizosa, E.; Molero-Río, C.; and Romero Morales, D. 2021. Mathematical optimization in classification and regression trees. *Transactions in Operations Research*, 29(1): 5–33.

Costa, V. G.; and Pedreira, C. E. 2023. Recent advances in decision trees: An updated survey. *Artificial Intelligence Review*, 56(5): 4765–4800.

Demirovic, E.; Lukina, A.; Hebrard, E.; Chan, J.; Bailey, J.; Leckie, C.; Ramamohanarao, K.; and Stuckey, P. J. 2022. MurTree: Optimal Decision Trees via Dynamic Programming and Search. *Journal of Machine Learning Research*, 23: 26:1–26:47.

Eiben, E.; Ordyniak, S.; Paesani, G.; and Szeider, S. 2023. Learning Small Decision Trees with Large Domain. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence (IJCAI '23)*, 3184–3192. ijcai.org.

Fayyad, U. M.; and Irani, K. B. 1990. What Should Be Minimized in a Decision Tree? In *Proceedings of the 8th Conference on Artificial Intelligence (AAAI '90)*, 749–754. AAAI Press / The MIT Press.

Frank, E.; Hall, M. A.; and Witten, I. H. 2016. *Data mining: practical machine learning tools and techniques, 4th Edition*. Morgan Kaufmann, Elsevier.

Gahlawat, H.; and Zehavi, M. 2024. Learning Small Decision Trees with Few Outliers: A Parameterized Perspective. In *Proceedings of the 38th Conference on Artificial Intelligence (AAAI '24)*, 12100–12108. AAAI Press.

Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual.

Hu, H.; Siala, M.; Hebrard, E.; and Huguet, M. 2020. Learning Optimal Decision Trees with MaxSAT and its Integration in AdaBoost. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, 1170–1176. ijcai.org.

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

Janota, M.; and Morgado, A. 2020. SAT-Based Encodings for Optimal Decision Trees with Explicit Paths. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178, 501–518. Springer.

Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, 85–103. Plenum Press, New York.

Kobourov, S. G.; Löffler, M.; Montecchiani, F.; Pilipczuk, M.; Rutter, I.; Seidel, R.; Sorge, M.; and Wulms, J. 2023. The Influence of Dimensions on the Complexity of Computing Decision Trees. In *Proceedings of the 37th Conference on Artificial Intelligence (AAAI '23)*, 8343–8350. AAAI Press.

Komusiewicz, C.; Kunz, P.; Sommer, F.; and Sorge, M. 2023. On Computing Optimal Tree Ensembles. In *Proceedings of the International Conference on Machine Learning (ICML '23)*, volume 202 of *Proceedings of Machine Learning Research*, 17364–17374. PMLR.

Molnar, C. 2020. *Interpretable Machine Learning*. Independently published.

Moshkovitz, M.; Dasgupta, S.; Rashtchian, C.; and Frost, N. 2020. Explainable k-Means and k-Medians Clustering. In *Proceedings of the 37th International Conference on Machine Learning (ICML '20)*, 7055–7065.

Narodytska, N.; Ignatiev, A.; Pereira, F.; and Marques-Silva, J. 2018. Learning Optimal Decision Trees with SAT. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, 1362–1368. ijcai.org.

Ordyniak, S.; Paesani, G.; Rychlicki, M.; and Szeider, S. 2024. A General Theoretical Framework for Learning Smallest Interpretable Models. In *Proceedings of the 38th Conference on Artificial Intelligence (AAAI '24)*, 10662–10669. AAAI Press.

Ordyniak, S.; and Szeider, S. 2021. Parameterized Complexity of Small Decision Tree Learning. In *Proceedings of the 35th Conference on Artificial Intelligence (AAAI '21)*, 6454–6462. AAAI Press.

Romano, J. D.; Le, T. T.; Cava, W. G. L.; Gregg, J. T.; Goldberg, D. J.; Chakraborty, P.; Ray, N. L.; Himmelstein, D. S.; Fu, W.; and Moore, J. H. 2022. PMLB v1.0: an open-source dataset collection for benchmarking machine learning methods. *Bioinformatics*, 38(3): 878–880.

Ruggieri, S. 2004. YaDT: Yet another Decision Tree Builder. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '04)*, 260–265. IEEE Computer Society.

Ruggieri, S. 2019. Complete Search for Feature Selection in Decision Trees. *Journal of Machine Learning Research*, 20: 104:1–104:34.

Savnik, I. 2013. Index Data Structure for Fast Subset and Superset Queries. In *Proceedings of the International Cross-Domain Conference on Availability, Reliability, and Security in Information Systems (CD-ARES '13)*, volume 8127 of *Lecture Notes in Computer Science*, 134–148. Springer.

Staus, L. P. 2024. *Efficient Algorithms for Learning Minimal Decision Trees*. Master's thesis, Philipps-Universität Marburg. https://www.fmi.uni-jena.de/fmi_femedia/33286/ma-staus-pdf.pdf.