Philipps-Universität Marburg

Fachbereich Mathematik und Informatik

Efficient Algorithms for Learning Minimal Decision Trees

Masterarbeit

zur Erlangung des akademischen Grades eines

Master of Science

im Studiengang

Informatik

vorgelegt von

Luca Pascal Staus

am 03.01.2024

Betreuer:
Prof. Dr. Bernhard Seeger (Erstgutachter)
Prof. Dr. Christian Komusiewicz (Zweitgutachter)
Dr. Frank Sommer
Dr. Manuel Sorge

Abstract

Decision trees are a simple and interpretable way to classify data using a binary tree structure. The size of the decision tree is the number of inner vertices and a lot of decision tree problems try to find decision trees with a small size. One such problem is the Minimum Decision Tree Size (MDTS) problem. MDTS is an NP-hard problem where, for a given dataset, we try to find the smallest size s such that there is a decision tree of size at most s that correctly classifies all examples in the dataset.

In this work we will present a search tree algorithm that solves MDTS. We will start with a basic algorithm that recursively extends decision trees by adding a new inner vertex and a new leaf. This algorithm is a special case of the witness tree algorithm by Komusiewicz et al. [1]. We will then present multiple running time improvements for this algorithm.

We will test our algorithm with all improvements on 700 randomly sampled instances from 35 datasets that are part of the Penn Machine Learning Benchmarks [2]. We then compare these results with the state of the art algorithms for MDTS by Narodytska et al. [3] and Janota et al. [4]. We will show that our algorithm is able to solve roughly 25% more instances than the state of the art algorithms.

Contents

1 Introduction

Decision trees have been a popular model for classifiying multi-dimensional data ever since they were first introduced by Morgan et al. [5]. Usually they are rooted binary trees where each vertex is either an inner vertex with a left and right child or a leaf. Additionally each leaf is labeled with a class and each inner vertex in the tree is labeled with a cut (i, t) where i is a dimension of the data and t is a value in that dimension called a threshold. The size of the tree is the number of inner vertices and the depth of the tree is the number of edges of the longest path from the root to any leaf. Figure 1a shows an example dataset and Figure 1b shows an example decision tree for that dataset.

A decision tree classifies an example by assigning it to a specific leaf which then represents the class that is assigned to the example. To do this, each inner vertex with a cut (i, t) represents a logical test that assigns all examples to its left subtree that have a value in dimension i that is less than or equal to the threshold t. All other examples get assigned to the right subtree. By starting at the root each example ends up in exactly one leaf. The example a from the dataset in Figure 1a would for example be assigned to the leaf E by the decision tree in Figure 1b. This is because in dimension d_1 the value of a is less than or equal to 1 and in dimension d_2 the value of a is bigger than 0.

The example in Figure 1 also illustrates the interpretability of decision trees. For example, by just looking at the tree in Figure 1b it is easy to understand why the tree would assign the class blue to an example. However, if the tree was much bigger with hundreds of inner vertices it would no longer be as easy to interpret. Additionally, large decision trees could lead to overfitting which can decrease their accuracy on examples that are not part of the training dataset.

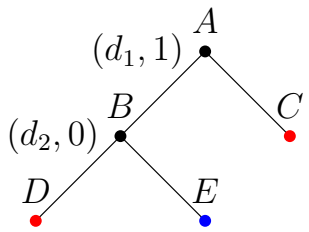
Because of this, a lot of decision tree research is focused on finding small trees. Traditionally, this was done using heuristics such as the CART algorithm by Breiman et al. [6]. This is because the problem of finding the smallest decision tree that correctly classifies a training dataset has been shown to be NP-hard [7].

In the last ten years, however, more researchers have developed algorithms that can find optimal decision trees [8, 9, 4, 10]. Not all of these algorithms solve the exact same problems though. Some algorithms try to find a decision tree that minimizes the number of incorrectly classified training examples while the tree is not allowed to exceed a certain size or depth. These algorithms usually use Mathematical Programming [11, 9] or they are search tree algorithms that use different techniques to prune the search tree [12, 10]. Other algorithms try to find the smallest tree that correctly classifies all examples. They do this by either minimizing the size of the tree [3, 4] or the depth of the tree [13, 14]. These algorithms generally use SAT formulations to solve the problem using generic SAT-solvers. For a more complete overview of the last decade of decision tree research, we refer to a review by Costa et al. [8].

In this work we will focus on the problem of finding the smallest decision tree that correctly classifies all examples in a training dataset. We will call this problem Minimum Decision Tree Size (MDTS). The most recent algorithm that solves MDTS is the SAT-based algorithm by Janota et al. [4]. In addition to presenting a novel SAT-encoding, Janota et al. also propose the idea of splitting the search-space by different

	d_1	d_2	class
a	0	1	blue
b	1	1	blue
c	1	0	red
d	2	2	red

(a) An example dataset with $n = 4$ examples, $d = 2$ dimensions and the set of class symbols $\Sigma = \{\text{blue}, \text{red}\}$.



(b) An example witness tree with the two inner vertices A and B and the three leafs C, D and E. A has the cut $(d_1, 1)$ and B has the cut $(d_2, 2)$.

Figure 1: Example dataset and witness tree.

tree topologies and calling the SAT-solver once for each topology. They then used this idea not just to improve their own encoding but also a different encoding by Narodytska et al. [3].

In contrast to these SAT-based algorithms, we will present a search tree algorithm that solves MDTS. To do this we first solve the decision version of MDTS called Decision Tree Size (DTS) where we need to decide if a decision tree with a maximum size s exists that correctly classifies all training examples. We can then solve MDTS by linearly increasing the maximum size by one until we find a tree that correctly classifies the data.

The base version of our algorithm that solves DTS is a special case of the witness tree algorithm proposed by Komusiewicz et al. [1]. Their algorithm solves a generalized version of DTS called Tree Ensemble Size (TES) where an ensemble of decision trees needs to be found that correctly classifies all examples while the sum of the sizes of all trees in the ensemble is not bigger than a given maximum size S. Classification of an example e by a tree ensemble works by first classifying e with each individual tree. The class assigned to e by the ensemble is then the class that was assigned to e by the majority of individual trees.

Next, we will describe how the special case of the witness tree algorithm with ensemble size one works. Generally the algorithm is a search tree algorithm where each node in the search tree represents a valid decision tree. If the decision tree of the current node incorrectly classifies at least one example e then the algorithm goes through all possible ways of adding a new inner vertex and a new leaf to the tree such that e gets assigned to the new leaf. The class of e is then assigned to the new leaf so that e is correctly classified by the new decision tree. This procedure is called a one-step-refinement.

To make sure that e does not get incorrectly classified again by other subsequent one-step-refinements, an extension of a decision tree called a witness tree is used. In a witness tree each leaf is labeled with an example called a witness with the restriction that the witness always has to be assigned to this leaf by the tree. That means, when a one-step-refinement is performed, the dirty example e becomes the witness of the newly added leaf.

Overall the algorithm now does the following: It starts with a witness tree that has one leaf and no inner vertices. An arbitrary witness is chosen for this leaf and the class of the leaf becomes the class of the witness. Then, as long as there is at least one example e that is not correctly classified by the current tree and as long as the size of the tree is not bigger or equal to the maximum size s, the algorithm recursively applies all possible one-step-refinements that assign e to the new leaf and do not change the leafs of existing witnesses. The algorithm terminates when a tree with maximum size s is found that correctly classifies all examples or if the algorithm has discovered all trees that can be discovered this way.

Komusiewicz et al. [1] used this witness tree algorithm to prove that TES has a running time bound of $O((6\delta DS)^S \cdot S\ell n)$ where δ is the maximum number of dimensions in which two examples differ, D the maximum number of unique values in a dimension, S the maximum size of the trees in the ensemble, ℓ the number of trees in the ensemble and n the number of training examples. For the special case with $\ell = 1$ this running time bound becomes $O((\delta Ds)^s \cdot sn)$.

To make this special case of the witness tree algorithm viable in practice we present several improvements for the base algorithm in Sections 4 - 7. We will show that these improvements will significantly speed up the algorithm to the point where it performs significantly better than the state of the art algorithms for MDTS.

The rest of this work is structured in the following way. In Section 2, we will formally define decision trees and the associated notation. In Section 3, we will give a more in-depth explanation of how the special case of the witness tree algorithm works. In Sections 4 - 7 we will introduce several improvements that speed up the algorithm. Finally, in Section 8 we will compare our algorithm with the SAT-based algorithms by Narodytska et al. [3] and Janota et al. [4].

2 Preliminaries

We define $[n] := \{1, 2, \ldots, n\}$ for $n \in \mathbb{N}$. 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 will always assume $\Sigma = \{\text{blue}, \text{red}\}$ since we never consider more than two classes. A *dataset* with classes Σ is a tuple (E, λ) where $E \subseteq \mathbb{R}^d$ is a set of *examples* and $\lambda : E \to \Sigma$ is a map that maps each example to a class. From now on we assume that some dataset (E, λ) is always given with n being the number of examples and d the number of dimensions in that dataset. We also define δ as the maximum number of dimensions in which two examples differ and D as the maximum number of unique values in a dimension. For each dimension $i \in [d]$ we define $\text{Thr}(i)$ as the 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 $t \in \text{Thr}(i)$ with $e_1[i] \leq t < e_2[i]$. There are of course multiple sets with this property but for our purposes any set with this property is fine. A *cut* is a pair (i, t) where $i \in [d]$ is a dimension and $t \in \text{Thr}(i)$ is a threshold. The set of all cuts is $\text{Cuts}(E)$. We define the *left side* of a cut with respect to $E' \subseteq E$ as $E'[\leq (i, t)] := \{e \in E' \mid e[i] \leq t\}$. Similarly we define the *right side* of a cut with respect to E' as $E'[> (i, t)] := \{e \in E' \mid e[i] > t\}$.

A *decision tree* is a tuple $D = (T, \text{cut}, \text{cla})$ where T is a sorted and rooted binary tree with vertex set $V(D)$, $\text{cut} : V(D) \to \text{Cuts}(E)$ maps every inner vertex to a cut and $\text{cla} : V(D) \to \Sigma$ maps each leaf to a class. For each vertex $v \in V(D)$ we define a set $E[D, v] \subseteq E$ of examples that are *assigned* to v. If v is the root of D we simply define $E[D, v] := E$. Otherwise v has a parent vertex p and we define $E[D, v] := E[D, v][\leq \text{cut}(p)]$ if v is the left child of p and $E[D, v] := E[D, v][> \text{cut}(p)]$ if v is the right child of p. If D is clear we just write $E[v]$. With this definition each example $e \in E$ is assigned to exactly one leaf ℓ. We say that ℓ is the leaf of e in D and write $\text{leaf}(D, e) := \ell$ or just $\text{leaf}(e)$ if D is clear. An example $e \in E$ is dirty in a decision tree T if we have $\lambda(e) \neq \text{cla}(\ell)$ with ℓ being the leaf of e. The set of all dirty examples in T is $\text{Dirty}(T)$. We say that a decision tree classifies (E, λ) if the class of every example $e \in E$ matches the class of its leaf, i.e. we have $\lambda(e) = \text{cla}(\text{leaf}(e))$ for all $e \in E$. In this case we call D *correct*.

Now we can properly define the two problems we mentioned in Section 1.

Decision Tree Size (DTS)

Instance: A training data set (E, λ) and a size bound s.
Question: Is there a decision tree of size at most s that classifies (E, λ)?

Minimum Decision Tree Size (MDTS)

Instance: A training data set (E, λ).
Task: Find the smallest s such that there is a decision tree of size at most s that classifies (E, λ).

3 Base Algorithm and Experimental Setup

In this section we will explain in more detail how the base version of our algorithm works. As mentioned above our algorithm is a special case of the witness tree algorithm developed by Komusiewicz et al. [1] where the ensemble size is one. That means the correctness of this special case directly follows from the correctness of the witness tree algorithm.

Before we can properly explain how the algorithm works we first need to introduce some important definitions. We start by defining a *witness tree* as a tuple $W = (T, \text{cut}, \text{cla}, \text{wit})$ where $(T, \text{cut}, \text{cla})$ is a decision tree and $\text{wit} : V(T) \to E$ is a map that maps each leaf ℓ to an example $e \in E[\ell]$ such that the class of e matches the class of ℓ. We call these examples *witnesses* of W. Additionally, since a witness tree is just an extension of a decision tree, any definition and notation we introduced in Section 2 that involves a decision tree works the same way for witness trees.

Next we need a way to extend a witness tree by adding a new inner vertex and a new leaf. For this we define one-step-refinements. A *one-step-refinement* of a witness tree W is a tuple (v, i, t, e) where $v \in V(W)$ is some vertex in W, (i, t) is a cut in $\text{Cuts}(E)$ and $e \in E[W, v]$ is an example. We define $\text{RefAll}(W)$ as the set of all possible one-step-refinements for W. Applying a one-step-refinement $r = (v, i, t, e)$ to a witness tree W works in the following way.

First we subdivide the edge from v to its parent p by adding a new inner vertex u. Now, p is the parent of u and u is the parent of v. If v was previously the left child of p then u is now the new left child. Similarly, if v was previously the right child of p then u is now the new right child. It is of course possible that v is the root of the tree and therefore does not have a parent p. In that case u becomes the new root.

Next we add a new leaf ℓ as the second child of u with v being the first child. To determine which vertex is the left child and which vertex is the right child we look at the the example e and the cut (i, t). We want e to be assigned to the new leaf ℓ. That means if e is on the left side of (i, t), i.e. $e \in E[\leq (i, t)]$, we make ℓ the left child of u. Otherwise we make ℓ the right child of u. Finally we set the cut of u to (i, t), the class of ℓ to the class of e and the witness of ℓ to e.

In this way we create a new witness tree R and we write $W \xrightarrow{r} R$ to represent that R was created by applying r to W. If we apply a sequence of one-step-refinements $r_1, \ldots, r_n$ to create R we write $W \xrightarrow{r_1,\ldots,r_n} R$. Finally we define $\text{Ref}(W) \subseteq \text{RefAll}(W)$ as the set of all one-step-refinements $r = (v, i, t, e)$ of W such that $e \in \text{Dirty}(W)$ is dirty in W 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.

With these definitions we can now explain how the algorithm works. The algorithm starts with a witness tree that just contains one leaf and no inner vertices. We choose an arbitrary witness for this leaf and set the class of this leaf to the class of the witness. The algorithm then chooses a dirty example e and iterates over all one-step-refinements where e is the dirty example. The idea is that since e is currently dirty, we need to assign e to a new leaf that has the same class as e if we want to correctly classify all examples. The one-step-refinements with e as the dirty example do this by making e

Algorithm 1 Base Witness Tree Algorithm

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

```
1: function Refine(W, (E, λ), s)
2:     if W classifies (E, λ) then
3:         return W
4:     if size of W is equal to s then
5:         return ⊥
6:     e ← some dirty example from Dirty(W)
7:     for all r = (v, i, t, e) ∈ Ref(W) do
8:         Apply r to W to create the new witness tree R
9:         R' ← Refine(R, (E, λ), s)
10:        if R' ≠ ⊥ then
11:            return R'
12:    return ⊥
```

the witness of the leaf that they add to the tree and assigning the class of e to that leaf.

Next the algorithm applies these one-step-refinements to the current tree and then recursively calls itself for each new tree that is created in this way. The recursion either ends when the current tree classifies the data or when the tree has reached the maximum size s and does not classify the data. Algorithm 1 shows the pseudocode.

In Line 2 we first check if the witness tree already classifies the data. If that is not the case we check in Line 4 wether the tree has reached the maximum size s. In Line 6 we choose an arbitrary dirty example e and in Line 7 we iterate over every one-step-refinement in $\mathrm{Ref}(W)$ that uses e as the dirty example. Next we apply the current one-step-refinement to W and recursively call the algorithm with the new witness tree in Lines 8 and 9. In Line 10 we check if the recursive call has found a solution or not. Lastly if no solution could be found for any of the one-step-refinements we return $\perp$ in Line 12.

Running the algorithm creates a search tree where each node N represents a call of Refine. To avoid confusion we will always refer to the vertices of the search tree as nodes. We will use $\mathrm{Tree}(N)$ to refer to the witness tree W that Refine is called with and we will use $\mathrm{ex}(N)$ to refer to the dirty example that is chosen in Line 6 of the pseudo code. The nodes created by calling Refine in Line 9 are the children of the current node N.

We now explain the specific order in which the algorithm iterates over the one-step-refinements in Line 7. This ordering will also induce an ordering on the children of the current node N. First, the algorithm chooses the vertex v of the one-step-refinement by iterating over the vertices on the leaf to root path starting at the leaf ℓ of e. Next, the algorithm chooses the dimension i by iterating over all dimensions from 1 to d. Lastly, the algorithm chooses the threshold t by iterating over all thresholds in $\mathrm{Thr}(i)$ that are between $e[i]$ and $\mathrm{wit}(\ell)[i]$. The algorithm always starts with the thresholds that are

closest to $e[i]$ regardless of whether $e[i]$ is bigger than $\text{wit}(\ell)[i]$ or smaller. Some of those thresholds may change the leaf of some witnesses. The algorithm skips such thresholds.

3.1 Dirty Example Priority

We currently have not specified how exactly the algorithm chooses the dirty example in Line 6. This is because the example that we choose does not matter for the correctness of the algorithm. However, this also allows us to introduce the first improvement to the algorithm where we try to choose the dirty example such that the size of the search tree is as small as possible.

The motivation for this is the running time bound $O((\delta DS)^S \cdot Sn)$ for the algorithm that we previously mentioned in Section 1. To obtain this running time bound, Komusievicz et al. used δDS as an upper bound on the number of one-step-refinements that the algorithm loops through in Line 7. This shows that reducing the number of one-step-refinements that the algorithm loops through can have a large effect on the running time of the algorithm.

One way we can do this is to simply choose the dirty example e such that the number of one-step-refinements is as small as possible. However calculating this for every dirty example every time Refine is called would take too long. Instead we recalculate this number for an example e only when it is assigned to a new leaf. This can of course lead to these numbers being inaccurate if many one-step-refinements are performed on the tree that do not change the leaf of e. However our preliminary experiments have shown that this is a good tradeoff.

We can use the same idea to choose the witness of the initial leaf and the dirty example that is chosen in the first call of Refine. These two examples determine how many one-step-refinements the algorithm has to go through in the initial call of Refine. That means before the algorithm starts we can calculate the pair of examples with different classes that minimizes the number of one-step-refinements, that is, the number of cuts seperating them.

3.2 Solving MDTS

Next, we present a way to use Algorithm 1 to solve MDTS. The goal is to find an s such that Algorithm 1 returns a decision tree when called with s and $\perp$ when called with $s - 1$. To do this we can start by calculating a lower bound for the value of s. We then linearly increase s by one each time the algorithm returns $\perp$. This way we find the optimal value of s while never calling the algorithm for an s that is bigger than necessary.

For the lower bound we will initially just use $s = 1$. In Section 5 we will introduce a lower bound that is a lot better than this trivial lower bound.

Table 1: Overview of the datasets we used for our experiments including their name, number of examples n, number of dimensions d, number of total cuts c, maximum number of dimensions in which two examples differ δ and the maximum number of unique values in a dimension D

name	n	d	c	δ	D
appendicitis	106	7	523	7	99
australian	690	18	1155	16	350
auto	202	52	961	31	184
backache	180	55	469	26	180
biomed	209	14	735	9	191
breast-cancer	266	31	40	15	11
bupa	341	5	307	5	94
cars	392	12	704	9	346
cleve	302	27	390	18	152
cleveland	303	27	391	18	152
cleveland-nominal	130	17	17	11	2
cloud	108	7	585	7	108
colic	357	75	408	36	85
contraceptive	1358	21	66	13	34
dermatology	366	129	188	57	61
diabetes	768	8	1246	8	517
ecoli	327	7	351	6	81
glass	204	9	894	9	172
glass2	162	9	709	9	136
haberman	283	3	89	3	49
hayes-roth	84	15	15	8	2
heart-c	302	27	390	18	152
heart-h	293	29	325	19	154
heart-statlog	270	25	376	18	144
hepatitis	155	39	355	28	85
hungarian	293	29	325	19	154
lupus	86	3	126	3	75
lymphography	148	50	50	26	2
molecular_biology_promoters	106	228	228	104	2
new-thyroid	215	5	329	5	100
postoperative-patient-data	72	22	22	14	2
schizo	340	14	2218	14	203
soybean	622	133	133	68	2
spect	219	22	22	22	2
tae	106	5	96	5	46

3.3 Experimental Setup

In the following sections we will present several improvements to Algorithm 1. In order to show the effect of these improvements, we will show the results of experiments where we only use the improvements we have shown up to that point and the new improvement. We will then compare those results to the results of the previous section. After we have presented all of the improvements, we will compare the final version of our algorithm with the algorithms presented by Janota et al. [4] that also solve MDTS. In this section we will explain the details of how we performed those experiments.

Each of our experiments was performed on a single thread of an Intel(R) Xeon(R) Silver 4116 CPU with 2.1 GHz, 12 CPUs, 24 threads, and 128 GB RAM running Java openjdk 17.0.4. We implemented the algorithm in Kotlin and the code is available on Github at `https://github.com/LucaStaus/master-thesis-code`.

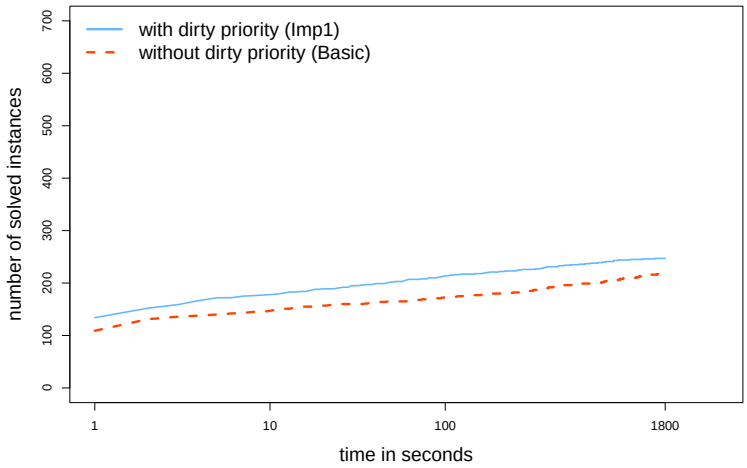


Figure 2: Comparison of the algorithm with and without dirty example priority.

For our experiments, we used 35 datasets that were also used by Janota et al. [4] for their experiments. The datasets are part of the Penn Machine Learning Benchmarks [2] and can all be found on this website `https://epistasislab.github.io/pmlb/index.html`. Table 1 shows an overview of the datasets.

Before we could use the datasets for our experiments we had to modify them slightly as follows: First we needed to replace categorical dimensions since our algorithm does not support such dimensions. We did that by creating a new binary dimension for each category. In each of those dimensions, we assigned the value 1 to an example if it has the category of that dimension and 0 if not. We also had to deal with some datasets having more than two classes. In that case we simply found the most common class and turned an example into a red example if it had that class. Otherwise, we turned it into a blue example. Lastly, there were some datasets with pairs of examples that had the same values in all dimensions but had different classes. We simply removed one example from each such pair.

Similar to Janota et al. [4], we randomly sampled multiple subsets of the examples from each dataset. Specifically, for each dataset we chose 10 random subsets with 20% of the examples and 10 random subsets with 50% of the examples. This means we ran our experiments on a total of 700 instances. Additionally, we ran each instance with a timeout of 30 minutes.

3.4 Evaluation

In this section, we evaluate the base version of our algorithm and show the effect of dirty example priority. To show these results we use Figure 2. It shows the total number of solved instances over time for each version of the algorithm. The red dashed line represents Algorithm 1 without dirty example priority and the blue line represents Algorithm 1 with dirty example priority. We will call these two versions *Basic* and *Imp1*

respectively.

In total, Basic solved 219 of the 700 instances. Imp1 was able to solve the same 219 instances plus an additional 28 instances. Out of the 219 instances that were solved by both algorithms, only 10 instances could be solved more than one second faster with Basic than with Imp1. Imp1 also significantly decreases the size of the search trees. On average, the search trees of Imp1 had 52% less nodes than the search trees of the same instances solved by Basic. This shows that the dirty example priority had the desired effect of reducing the number of children in each search tree node. Overall, we can conclude that dirty example priority is an improvement which almost always speeds up the algorithm substantially.

4 Data Reduction

In this section, we will present some techniques for reducing the input data. The idea is that we can sometimes transform a dataset into a smaller dataset without changing the solution to any instance of DTS. Here, smaller can refer to the number of examples, the number of dimensions, or the number of cuts. To formalize this idea we introduce reduction rules with the following definition.

Definition 4.1. A *reduction rule* is a function r that maps any dataset (E, λ) to another dataset (E', λ'). We call r *correct* if there is a decision tree of size s that correctly classifies (E, λ), if and only if there is a decision tree of size s that correctly classifies (E', λ').

In the following subsections we will introduce five reduction rules that decrease the size of the dataset. To demonstrate how they work we will use the dataset in Table 2 as an example.

4.1 Removing Examples and Dimensions

The first two reduction rules are very simple. We can use them to remove uneccessary examples or dimensions.

Remove Duplicate Example Rule:
Let (E, λ) ba a dataset and let $e_1, e_2 \in E$ be a pair of examples. If e_1 and e_2 have the same value in all dimensions, then remove e_1.

Remove Dimension Rule:
Let (E, λ) be a dataset and let i be a dimension. If all examples in i have the same value, then remove dimension i.

The first rule is clearly correct because two examples with the same values in all dimensions have the same class and would always end up in the same leaf of any decision tree. Removing one of them therefore does not change wether a decision tree is correct or not.

The second rule is clearly correct because a dimension where all examples have the same value does not have any cuts. That means a decision tree will never use this dimension anyway.

4.2 Equivalent Cuts

The third reduction rule is called the *Equivalent Cuts Rule*. The idea of this rule is to remove cuts from the dataset that are equivalent to other cuts. To better understand what it means for two cuts to be equivalent we can look at the two cuts $(d_1, 1)$ and $(d_2, 1)$ in Table 2. Their left sides $E[\leq (d_1, 1)]$ and $E[\leq (d_2, 1)]$ are both equal to $\{a, b\}$. That means if we replace the cut $(d_1, 1)$ in a decision tree with the cut $(d_2, 1)$ no example would be assigned to a different leaf.

Table 2: Example Dataset for Section 4.

	d_1	d_2	d_3	class
a	0	1	0	red
b	1	0	0	red
c	2	2	2	blue
d	3	2	1	red

With this we can define an equivalence relation over the set of all cuts Cuts(E). Two cuts (i_1, t_1), $(i_2, t_2) \in \text{Cuts}(E)$ are *equivalent* if $E[\leq (i_1, t_1)] = E[\leq (i_2, t_2)]$. We can now use this relation to define the Equivalent Cuts Rule:

Equivalent Cuts Rule:

Let (E, λ) be a dataset and let $(i_1, t_1), (i_2, t_2) \in \text{Cuts}(E)$ be two cuts. If (i_1, t_1) and (i_2, t_2) are equivalent, then remove (i_1, t_1).

Next we will explain how removing a cut (i, t) from a dataset works. To remove this cut, we need to remove the thresholds t from $\text{Thr}(i)$. According to the definition of $\text{Thr}(i)$ there must be at least one pair of examples $e_1, e_2 \in E$ such that t is the only threshold in dimension i with $e_1[i] \leq t < e_2[i]$. By setting the value in dimension i of the examples in every pair with this property to t, we remove t from $\text{Thr}(i)$. At the same time this transformation ensures that the left and right side of every other cut does not change.

If for example we want to remove the cut $(d_1, 1)$ from the dataset in Table 2 we would find that the only pair of examples that has the property mentioned above is the pair (b, c). All other pairs can also be split by the cuts $(d_1, 0)$ or $(d_1, 2)$. That means we now set the values of b and c in dimension d_1 to 1.

Now we just need to show that the Equivalent Cuts Rule is correct.

Lemma 4.1. *The Equivalent Cuts Rule is correct.*

Proof. Let (E, λ) be the original dataset and (E', λ') be the dataset that was created by the equivalent cuts rule. If a decision tree correctly classifies (E', λ') then it also correctly classifies (E, λ') since the rule only removes cuts. If a decision tree correctly classifies (E, λ) we can replace all cuts that were removed by the rule with the one cut that was not removed from their equivalence class. This creates a tree with the same size that correctly classifies (E', λ'). □

4.3 Reducing the size of Dimensions

With the fourth reduction rule, we want to remove the extreme values in some dimensions. For this let us again look at dimension d_1 in Table 2. The left side $E[\leq (d_1, 0)]$ of the cut $(d_1, 0)$ only contains a red example. The left side $E[\leq (d_1, 1)]$ of the cut $(d_1, 1)$ also only contains red examples while the right side contains fewer examples.

If one of these cuts is used in a decision tree that correctly classifies the data and does not contain empty leafs, then the left subtree of the vertex using these cuts would

Table 3: Example for merging the two dimensions d_2 and d_3.

(a) Before the merge.

	d_1	d_2	d_3	class
b	1	0	0	red
a	0	1	0	red
d	3	2	1	red
c	2	2	2	blue

(b) After the merge.

	d_1	d_4	class
b	1	0	red
a	0	1	red
d	3	2	red
c	2	3	blue

just be a single leaf with the class *red*. That means replacing the cut $(d_1, 0)$ with the cut $(d_1, 1)$ in such a tree would create a decision tree that has the same size and still correctly classifies the data.

This leads to the following reduction rule.

Dimension Reduction Rule:

Let (E, λ) be a dataset and let $(i, t_1), (i, t_2)$ with $t_1 < t_2$ be a pair of cuts. If all examples on the left side of both cuts have the same class, then remove (i, t_1). Similarly, if all examples on the right side of both cuts have the same class, then remove (i, t_2).

Removing a cut works the same way as it does with the Equivalent Cuts Rule.

Lemma 4.2. *The Dimension Reduction Rule is correct.*

Proof. Let (E, λ) be the original dataset and (E', λ') be the dataset that was created by the equivalent cuts rule. If a decision tree correctly classifies (E', λ'), then it also correctly classifies (E, λ') since the rule only removes cuts.

Let D be a decision tree that correctly classifies (E, λ). If the cut (i, t) of an inner vertex in D was removed and the threshold t is smaller than the smallest threshold t' of dimension i in (E', λ'), we can replace (i, t) with (i, t'). We can also do this replacement if t' is the biggest threshold of dimension i in (E', λ') and t is bigger than t'.

Without loss of generality, let us assume that t is smaller than t'. Since the rule removed (i, t) we know that all examples on the left sides of the cuts (i, t) and (i, t') must have the same class. Replacing (i, t) by (i, t') means the right side is still correctly classifies while the left side can be classified without any further cuts. This means there is a tree with the same size as D that correctly classifies (E', λ'). □

4.4 Merging Dimensions

With the last reduction rule we want to merge dimensions together in order to reduce the number of dimensions while keeping the number of different cuts the same. This may not seem useful at first but in Section 6 we will introduce an improvement of the algorithm that will benefit from this.

To better demonstrate what merging two dimensions means we can look at dimensions d_2 and d_3 in Table 2. If we now look at the examples in the order b, a, d, c as shown

Table 5: Overview of the datasets we used for our experiments including their name, number of examples n, number of dimensions d, number of total cuts c, maximum number of dimensions in which two examples differ δ and the maximum number of unique values in a dimension D. The columns n', d', c', δ' and D' show the values of the datasets after applying all reduction rules.

name	n	n'	d	d'	c	c'	δ	δ'	D	D'
appendicitis	106	106	7	7	523	**460**	7	7	99	**98**
australian	690	690	18	**16**	1155	**1119**	16	**15**	350	350
auto	202	202	52	**35**	961	**916**	31	**29**	184	**182**
backache	180	180	55	**50**	469	**429**	26	26	180	**151**
biomed	209	209	14	14	735	**577**	9	9	191	**125**
breast-cancer	266	266	31	**25**	40	40	15	15	11	11
bupa	341	341	5	5	307	**302**	5	5	94	94
cars	392	**388**	12	**11**	704	**531**	9	9	346	**266**
cleve	302	302	27	**25**	390	**382**	18	18	152	**151**
cleveland	303	303	27	**25**	391	**383**	18	18	152	**151**
cleveland-nominal	130	130	17	17	17	17	11	11	2	2
cloud	108	108	7	7	585	**555**	7	7	108	**100**
colic	357	357	75	**71**	408	**400**	36	36	85	**82**
contraceptive	1358	1358	21	21	66	**65**	13	13	34	34
dermatology	366	366	129	**101**	188	188	57	**53**	61	61
diabetes	768	768	8	8	1246	**1238**	8	8	517	**515**
ecoli	327	**326**	7	**5**	351	**233**	6	**5**	81	**59**
glass	204	204	9	9	894	**846**	9	9	172	**165**
glass2	162	162	9	9	709	**667**	9	9	136	**132**
haberman	283	283	3	3	89	**86**	3	3	49	**46**
hayes-roth	84	84	15	15	15	15	8	8	2	2
heart-c	302	302	27	**25**	390	**382**	18	18	152	**151**
heart-h	293	293	29	**22**	325	**318**	19	19	154	154
heart-statlog	270	270	25	**23**	376	**369**	18	18	144	**142**
hepatitis	155	155	39	**28**	355	**335**	28	**25**	85	85
hungarian	293	293	29	**22**	325	**318**	19	19	154	154
lupus	86	**79**	3	**2**	126	**78**	3	**2**	75	**53**
lymphography	148	148	50	**37**	50	50	26	**23**	**2**	3
molecular_biology_promoters	106	106	228	228	228	228	104	104	2	2
new-thyroid	215	**214**	5	5	329	**232**	5	5	100	**73**
postoperative-patient-data	72	72	22	**17**	22	22	14	14	**2**	3
schizo	340	340	14	14	2218	**2209**	14	14	203	203
soybean	622	622	133	**73**	133	**108**	68	**49**	**2**	7
spect	219	219	22	22	22	22	22	22	2	2
tae	106	106	5	5	96	**94**	5	5	46	**45**

in Table 4a we can see that their values in both dimensions never decrease. We can now create a new dimension d_4 with the values shown in Table 4b. We chose the values such that for each cut in dimension d_2 or d_3, there is an equivalent cut in d_4. This means we can now completely remove dimensions d_2 and d_3 and if any decision tree uses a cut in one of these dimensions, we can just replace it with the equivalent cut in d_4. This leads to the following rule.

Dimension Merge Rule:

Let (E, λ) be a dataset and let i_1, i_2 be a pair of dimensions. If there is an ordering of the examples such that their values never decrease in either dimension, then we can create a new dimension $i_{1,2}$ such that for each cut in dimensions i_1 and i_2, there is an equivalent cut in dimension $i_{1,2}$. We then remove i_1 and i_2.

We can generate the values of each example in the new dimension in the following

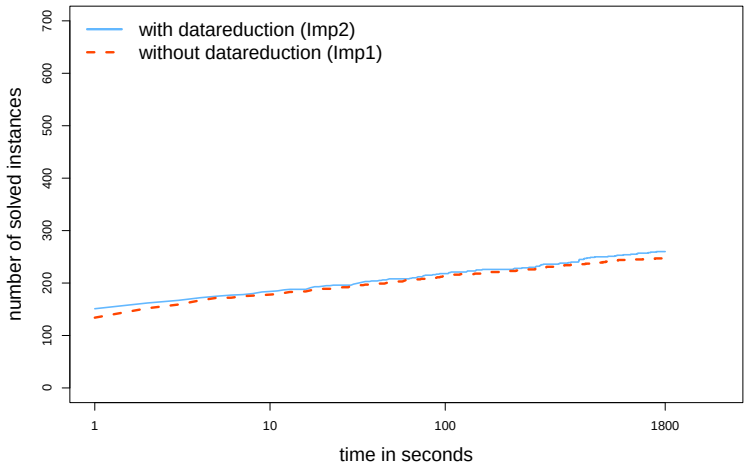


Figure 3: Comparison of the algorithm with and without datareduction.

way: We go through the examples in the order mentioned in the rule. We then start by assigning each example the value 0. Every time the value of the examples increases in one of the two original dimensions, we need a cut in the new dimension that has all the previous examples on its left side. To achieve this, we increase the value we assign to the remaining examples in the new dimension by 1.

Lemma 4.3. *The Dimension Merge Rule is correct.*

Proof. Let (E, λ) be the original dataset and (E', λ') be the dataset that was created by the dimension merge rule. After adding the new dimension to (E', λ'), there is an equivalent cut in this new dimension for each cut in the original two dimensions. The correctness of the Equivalent Cuts Rule shows that removing all cuts from the two dimensions that were merged together is correct. Removing these cuts leads to all examples having the same value in these two dimensions. The correctness of the Remove Dimension Rule shows that removing these two dimensions is also correct. □

4.5 Evaluation

We now evaluate the effectiveness of the above reduction rules. First, however, we need to define the order in which we use the reduction rules. This is important because applying the same rules to the same dataset in two different orders may lead to different results.

We start by removing as many cuts as possible. We do this by first using the Dimension Reduction Rule and then the Equivalent Cuts Rule. Next we use the Dimension Merge Rule to remove as many dimensions as possible. Finally we use the Remove Duplicate Examples Rule and the Remove Dimension Rule to clean up the data. It is important to note that if we say we use a certain rule, then that means we use the rule as many times as possible and not just once.

To give a general overview of how effective these rules are, we can apply them to the datasets in Table 1. The results of this are shown in Table 5.

Next, we will evaluate the effect that these reduction rules have on the running time of the algorithm. For this we will compare a version of our algorithm that uses all improvements we have shown so far including all reduction rules with a version that uses all improvements except the reduction rules. We will refer to these algorithms as *Imp2* and *Imp1*, respectively.

Figure 3 shows this comparison. Imp1 was able to solve 247 out of the 700 total instances. Imp2 solved 260 instances and was therefore able to solve slightly more instances than Imp1. There were however two instances that Imp1 could solve that were not solved by Imp2. Out of the 245 instances that were solved by both algorithms, there were only 12 where Imp2 took more than one second longer to solve them than Imp1.

This shows that, overall, Imp2 performed slightly better than Imp1. Moreover, the main advantage of these reduction rules is that they are a protection against datasets with a high level of redundancy.

5 Lower Bounds

In this section we will introduce two lower bounds that we can 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. The definition of Set Cover is as follows.

Set Cover
Instance: A universe U and a family S of subsets of U.
Task: Compute the size k of the smallest subset $S' \subseteq S$ such that the union of all sets in S' is equal to U.

We will 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 the current witness tree. We can then 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 $\bot$.

However, since Set Cover is NP-hard 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.

5.1 Improvement Lower Bound

The first lower bound is called the *Improvement Lower Bound (ImpLB)*. The idea behind this lower bound 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 will 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 5.1. Let W be a witness tree, $E' \subseteq \text{Dirty}(W)$ a set of dirty examples and $r \in \text{Ref}(W)$ a one-step-refinement with $W \xrightarrow{r} R$. Then we define

$$\text{imp}(W, E', r) := \{e' \in E' \mid e' \notin \text{Dirty}(R)\}$$

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

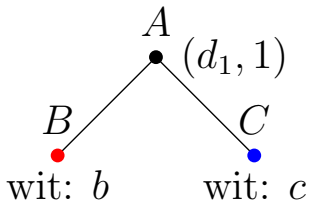
With this definition an imp set is basically a set of dirty examples that can all be correctly classified with just a single one-step-refinement. We can now create an instance of Set Cover by choosing $\text{Dirty}(W)$ as the universe and

$$I = \{\text{imp}(W, \text{Dirty}(W), r) \mid r \in \text{Ref}(W)\}$$

as the family of subsets. To demonstrate how an instance like this can look, consider the example in Figure 4. On the left it shows a dataset and on the right it shows a witness

	d_1	d_2	class
a	0	3	blue
b	1	2	red
c	2	2	blue
d	2	1	red
e	2	0	blue

(a) An example dataset with $n = 5$ and $d = 2$.



(b) An example witness tree. A has the cut $(d_1, 1)$, B has the witness b and C has the witness c.

Figure 4: Example dataset and witness tree.

tree W with one inner vertex A and two leafs B and C. The set $\text{Ref}(W)$ contains six one-step-refinements.

$$\text{Ref}(W) = \{(B, d_1, 0, a), (B, d_2, 2, a), (A, d_1, 0, a), (A, d_2, 2, a), (C, d_2, 1, d), (A, d_2, 1, d)\}$$

The imp sets $\text{imp}(W, \text{Dirty}(W), r)$ for these one-step-refinements look like this.

$$\{a\}, \{a\}, \{a\}, \{a\}, \{d\}, \{d\}$$

That means the Set Cover instance for this example has the universe $U = \{a, d\}$ and the set of subsets $I = \{\{a\}, \{d\}\}$.

5.1.1 Correctness Proof

To show that the solution of this instance is a lower bound as described above, we show that for any series of s one-step-refinements that correctly classify W there is a set $I' \subseteq I$ of size at most s such that the union of all sets in I' is equal to $\text{Dirty}(W)$. This is captured in Theorem 5.1.

Theorem 5.1. *Let $W := R_0$ be a witness tree and $(r_1, \ldots, r_s)$ a series of one-step-refinements with $R_{i-1} \xrightarrow{r_i} R_i$, $r_i \in \text{Ref}(R_{i-1})$ for $i \in [s]$ such that R_s classifies (E, λ). Then there must be a set $I \subseteq \text{Ref}(W)$, $|I| \leq s$, such that*

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

However to prove Theorem 5.1 we first need to prove Lemma 5.2. With this lemma we want to show the following: If R was created by applying a one-step-refinement to W then for any imp set S in R there is an imp set in W that is a superset of S. An important detail is that we calculate these imp sets with respect to the dirty examples in W. That means if there is a dirty example in R that was not dirty in W we will ignore it.

Lemma 5.2. *Let W be a witness tree, $E' \subseteq \text{Dirty}(W)$ a subset of the dirty examples in W, and $r \in \text{Ref}(W)$ a one-step-refinement with $W \xrightarrow{r} R$. Then, for each one-step-refinement $r_2 \in \text{Ref}(R)$ there exists a one-step-refinement $r_1 \in \text{Ref}(W)$, such that*

$$\text{imp}(R, E'', r_2) \subseteq \text{imp}(W, E', r_1)$$

with $E'' := E' \setminus \text{imp}(W, E', r)$.

Proof. Let $r = (v, i, t, e)$ and let $r_2 = (v', i', t', e') \in \text{Ref}(R)$ be some one-step-refinement for R. We now just need to find some one-step-refinement $r_1 \in \text{Ref}(W)$ such that $\text{imp}(R, E'', r_2) \subseteq \text{imp}(W, E', r_1)$. We can assume that $\text{imp}(R, E'', r_2) \neq \emptyset$, as otherwise $\text{imp}(R, E'', r_2) = \emptyset \subseteq \text{imp}(W, E', r_1)$ for any $r_1 \in \text{Ref}(W)$. By definition we know that $\text{imp}(R, E'', r_2) \subseteq E'' \subseteq E' \subseteq \text{Dirty}(W)$. Now we know there is some example $e'' \in \text{imp}(R, E'', r_2)$ which is dirty in W.

Let u be the inner node and ℓ the leaf that are added to W by r. We can now distinguish between three cases based on the vertex v'.

1. ($v' \in V(W)$): Here we choose $r_1 = (v', i', t', e'')$. Since e'' was correctly classified by r_2 we know that e'' must be in $E[R, v'] \subseteq E[W, v']$. We also know that r_1 does not change the leaf of any witnesses since $r_2 \in \text{Ref}(R)$. Therefore we have $r_1 \in \text{Ref}(W)$. Any example $d \in \text{imp}(R, E'', r_2)$ is a dirty example in W and contained in $E[W, v']$. Therefore r_1 correctly classifies d and we have $\text{imp}(R, E'', r_2) \subseteq \text{imp}(W, E', r_1)$.

2. ($v' = u$): Here we choose $r_1 = (v, i', t', e'')$. The proof for this case works as in the first case since we have $E[R, v'] = E[R, u] = E[W, v]$.

3. ($v' = \ell$): Here we choose $r_1 = (v, i, t, e'')$. We know that $e'' \in E[R, \ell] \subseteq E[W, v]$. We also know that r_1 does not change the leaf of any witnesses since $r \in \text{Ref}(W)$. Therefore we have $r_1 \in \text{Ref}(W)$. Any example $d \in \text{imp}(R, E'', r_2)$ was assigned to ℓ by the inner node u that was added by r. Since we also have $d \in E[W, v]$ we know that d will be assigned to the new leaf that is created by r_1. All examples in $\text{imp}(R, E'', r_2)$ have the same class which means $\lambda(e'') = \lambda(d)$. Therefore r_1 correctly classifies d and we have $\text{imp}(R, E'', r_2) \subseteq \text{imp}(W, E', r_1)$.

□

With this lemma we can now prove Theorem 5.1.

Proof for Theorem 5.1. Let $E_0 := \text{Dirty}(W)$ and $E_i := E_{i-1} \setminus \text{imp}(R_{i-1}, E_{i-1}, r_i)$ for all $i \in [s]$. By Lemma 5.2, for each $i \in [s]$ there is a one-step-refinement $r'_i \in \text{Ref}(W)$ with the following property:

$$\text{imp}(R_{i-1}, E_{i-1}, r_i) \subseteq \text{imp}(W, \text{Dirty}(W), r'_i). \tag{1}$$

Let $I = \{r'_i \mid i \in [s]\}$. Clearly $|I| \leq s$. Since R_s classifies (E, λ) and $E_s \subseteq \text{Dirty}(R_s)$ we must have $E_s = \emptyset$. This implies $\bigcup_{i \in [s]} \text{imp}(R_{i-1}, E_{i-1}, r_i) = \text{Dirty}(W)$. Due to Property (1) we now also know $\bigcup_{r \in I} \text{imp}(W, \text{Dirty}(W), r) = \text{Dirty}(W)$. □

5.1.2 Calculating the ImpLB

As mentioned above, we cannot simply calculate the solution for the Set Cover instance due to the NP-hardness of Set Cover. Instead, we calculate a lower bound for the Set Cover instance.

The idea is to not look at the content of each set, but to look at their sizes instead. Specifically this means we now try to find the smallest set of subsets such that the sum of their sizes is bigger or equal to the size of the universe. Clearly, this is a lower bound.

To calculate the ImpLB we now just need to look at each one-step-refinement $r \in \text{Ref}(W)$, calculate the size of $\text{imp}(W, \text{Dirty}(W), r)$, sort the sizes in descending order and then check how many are needed to reach a sum that is at least the size of $\text{Dirty}(W)$.

For the example in Figure 4 this would not make a difference since each subset in I has size one and there is no overlap. But let us instead assume we had the universe $U = \{a, b, c, d\}$ and the set of subsets $I = \{\{a, b\}, \{b, c\}, \{d\}\}$. The optimal solution for this instance of Set Cover would be 3 since we need all three sets to cover U. Our method turns I into a list of the sizes of each set sorted in descending order, giving $[2, 2, 1]$. We would then sum up the numbers from left to right until we reach the size of U. In this case we would need two numbers which is worse than the exact result. But for larger instances this method is much quicker to calculate since we do not need to look at the overlap between the sets. However there are some things we can do to speed up the calculation even more and improve the result at the same time.

First, we can use a data reduction technique for Set Cover instances. If a set A is a subset of a set B then we can remove A from our instance since it would always be better to choose B instead. Of course checking this subset relationship for every pair of imp sets in our instance every time we want to calculate the ImpLB would be inefficient. Instead, we just eliminate sets where we already know that they are a subset of a different set. We do this in the following way:

Consider two one-step-refinements $r_1 = (p, i, t, e)$ and $r_2 = (v, i, t, e)$ where p is the parent of v. Clearly $\text{imp}(W, \text{Dirty}(W), r_2)$ is a subset of $\text{imp}(W, \text{Dirty}(W), r_1)$ since both one-step-refinements use the same cut and $E[W, v]$ is a subset of $E[W, p]$. So when calculating the ImpLB we now only use one-step-refinements $r = (v, i, t, e)$ if the same one-step-refinement can not be applied at the parent p of v, i.e. when (p, i, t, e) is not in $\text{Ref}(W)$.

For the next improvement, consider the example in Figure 4 again. The two one-step-refinements $(B, d_1, 0, a)$, $(B, d_2, 2, a)$ are both applied to the same leaf B. If we now look at the sizes of their imp sets and add them up we get 2 since they both correctly classify the dirty example a. But we know that the set $E[B]$ only contains one dirty example. That means we can ignore one of the two one-step-refinements since we only need one to reach the number of dirty examples in that leaf. In this example this does not make a difference but in bigger instances we might be able to ignore very large sets in this way. Additionally, we can use this idea for any subtree and not just leafs.

Algorithm 2 ImpLB Algorithm

Input: A witness tree W and a training data set (E, λ).
Output: A lower bound for the number of one-step-refinements that are still required for W to classify (E, λ).

```
1: function CalcImpLB(W,(E, λ))
2:     for all v ∈ V(W) do P_v ← empty Priority Queue
3:     root ← root of W
4:     Recurse(W,(E, λ),root)
5:     return |P_root|
6: function Recurse(W,(E, λ),v)
7:     if v is an inner vertex then
8:         ℓ, r ← left child of v, right child of v
9:         Recurse(W,(E, λ),ℓ)
10:        Recurse(W,(E, λ),r)
11:        P_v.addAll(P_ℓ)
12:        P_v.addAll(P_r)
13:    p ← parent of v or null if v is the root of W
14:    for all r = (v, i, t, e) ∈ Ref(W) do
15:        if p = null ∨ (p, i, t, e) ∉ Ref(W) then
16:            P_v.add(|imp(W, Dirty(W), r)|)
17:    d ← number of dirty examples in the subtree of v
18:    while sum(P_v) − P_v.min() > d do
19:        P_v.removeMin()
```

5.1.3 Algorithm for ImpLB

Algorithm 2 shows the pseudo code for calculating the ImpLB as described above including the two improvements. The algorithm starts by initializing an empty priority queue for each vertex in the tree in Line 2. These are for storing and sorting the sizes of the imp sets. Next Recurse is called for the root of W. Recurse starts by calling itself for the left and right child of v and adding the content of their priority queues to the priority queue of v in Lines 7 through 12. This is of course only done if v is not a leaf. Next, in Line 14, the algorithm iterates over every one-step-refinement in $\text{Ref}(W)$ that can be performed at v. In Line 15 the algorithm checks that the same one-step-refinement can not be applied at the parent if a parent exists. This is the first improvement we mentioned above. The size of the imp set of r is then added to P_v in Line 16. Finally in Lines 17 – 19 we remove the smallest numbers from P_v until removing even one more would make the sum of the numbers in P_v smaller than the number of dirty examples in the subtree of v. This is the second improvement we mentioned above. The ImpLB is then just the size of P_{root} as can be seen in Line 5.

5.2 Pair Lower Bound

Next we define the *Pair Lower Bound (PairLB)*. For this lower bound we look at pairs of vertices with different classes. For a witness tree W, we define $\text{Pairs}(W)$ as the *set of all pairs* $\{e_1, e_2\} \subseteq E$ with $\lambda(e_1) \neq \lambda(e_2)$ and $\text{leaf}(e_1) = \text{leaf}(e_2)$, that is the set of all pairs of examples that are assigned to the same leaf but have different classes. Clearly each pair will always contain exactly one dirty example. For the example in Figure 4 this set would be $\text{Pairs}(W) = \{\{a, b\}, \{c, d\}, \{d, e\}\}$.

In a tree that correctly classifies the data this set must be empty since examples with different classes must be in different leafs. Because of this we can now define a lower bound that is similar to the ImpLB but instead of correctly classifying all dirty examples we try to split up all existing pairs. We use Definition 5.2 to capture this idea.

Definition 5.2. For some subset of pairs $P \subseteq \text{Pairs}(W)$ and a one-step-refinement $r \in \text{RefAll}(W)$ with $W \xrightarrow{r} R$, we define

$$\text{pairsplit}(W, P, r) := \{p \in P \mid p \notin \text{Pairs}(R)\}$$

as the *set of pairs that get split up by* r We call these sets the *pairsplit sets*.

Similar to the imp sets a pairsplit set is a subset of $\text{Pairs}(W)$ in which all pairs can be split up by a single one-step-refinement.

We can now create an instance of Set Cover by choosing $\text{Pairs}(W)$ as the universe and $P = \{\text{pairsplit}(W, \text{Pairs}(W), r) \mid r \in \text{RefAll}(W)\}$ as the family of subsets. Notice how we use the set of all one-step-refinements $\text{RefAll}(W)$ instead of the set $\text{Ref}(W)$ that only contains one-step-refinement that the algorithm is allowed to use. This is because this lower bound would not work if we used $\text{Ref}(W)$ instead of $\text{RefAll}(W)$. We can see this by looking at the example in Figure 4 again. If we construct the pairsplit sets for each one-step-refinement in $\text{Ref}(W)$ for this example we would get the following sets.

$$\{\{a, b\}\}, \{\{a, b\}\}, \{\{a, b\}\}, \{\{a, b\}\}, \{\{c, d\}\}, \{\{c, d\}\}.$$

Notice how none of these sets contain the pair $\{d, e\}$. This means if we only used the one-step-refinements in $\text{Ref}(W)$ to construct the Set Cover instance, it would have no solution. If we actually use all one-step-refinements in $\text{RefAll}(W)$ to construct the set of subsets P and we remove all duplicates, we would get $P = \{\{\{a, b\}\}, \{\{c, d\}\}, \{\{d, e\}\}\}$ since there are no one-step-refinements that can split more than one pair at a time.

5.2.1 Correctness Proof

To show that the solution of such an instance of Set Cover is a correct lower bound, we just need to show that for any series of s one-step-refinements that correctly classify W, there is a set $P' \subseteq P$ of size at most s such that the union of all sets in P' is equal to $\text{Pairs}(W)$. This is captured in the following theorem.

Theorem 5.3. *Let $W := R_0$ be a witness tree and $(r_1, \ldots, r_s)$ a series of one-step-refinements with $R_{i-1} \xrightarrow{r_i} R_i$, $r_i \in \text{RefAll}(R_{i-1})$ such that R_s classifies (E, λ). Then, there must be a set of at most s one-step-refinements $I \subseteq \text{RefAll}(W)$, $|I| \leq s$, such that*

$$\bigcup_{r \in I} \text{pairsplit}(W, \text{Pairs}(W), r) = \text{Pairs}(W).$$

Again similar to the ImpLB, we first need a lemma that has the same function as Lemma 5.2: If R was created by applying a one-step-refinement to W then for any pairsplit set in R, we want to show that there is a pairsplit set in W that is a superset of the pairsplit set in R. An important detail is that we calculate these pairsplit sets with respect to $\text{Pairs}(W)$. That means if there is a pair in $\text{Pairs}(R)$ that is not in $\text{Pairs}(W)$ we will ignore it.

Lemma 5.4. *Let W be a witness tree, $P \subseteq \text{Pairs}(W)$ a subset of the pairs in W and $r \in \text{Ref}(W)$ a one-step-refinement with $W \xrightarrow{r} R$. Then for each one-step-refinement $r_2 \in \text{RefAll}(R)$, there exists a one-step-refinement $r_1 \in \text{RefAll}(W)$, such that*

$$\text{pairsplit}(R, P', r_2) \subseteq \text{pairsplit}(W, P, r_1)$$

with $P' := P \setminus \text{pairsplit}(W, P, r)$.

Proof. Let $r_2 = (v, i, t, e) \in \text{RefAll}(R)$ be some one-step-refinement for R. We now just need to find some one-step-refinement $r_1 \in \text{RefAll}(W)$ such that $\text{pairsplit}(R, P', r_2) \subseteq \text{pairsplit}(W, P, r_1)$.

Since R was created by applying a one-step-refinement to W we have $\text{RefAll}(W) \subseteq \text{RefAll}(R)$. Because of this we can simply choose $r_1 = r_2$. We know that $P' \subseteq P$ and we know that pairs in P' can never be added to subtrees in R that they were not originally in before r was applied to W. Because of this we have $\text{pairsplit}(R, P', r_2) \subseteq \text{pairsplit}(W, P, r_1)$. □

Now we can prove Theorem 5.3. Due to the similarities in how we defined the ImpLB and the PairLB this proof works exactly the same as the proof for Theorem 5.1.

Proof for Theorem 5.3. Let $P_0 := \text{Pairs}(W)$ and $P_i := P_{i-1} \setminus \text{pairsplit}(R_{i-1}, P_{i-1}, r_i)$ for all $i \in [s]$. Due to Lemma 5.4 we know that for each $i \in [s]$ there is a one-step-refinement $r'_i \in \text{RefAll}(W)$ with the following property:

$$\text{pairsplit}(R_{i-1}, P_{i-1}, r_i) \subseteq \text{pairsplit}(W, \text{Pairs}(W), r'_i). \tag{2}$$

Let $I = \{r'_i \mid i \in [s]\}$. Clearly $|I| \leq s$. Since R_s classifies (E, λ) and $P_s \subseteq \text{Pairs}(R_s)$ we must have $P_s = \emptyset$. That implies $\bigcup_{i \in [s]} \text{pairsplit}(R_{i-1}, P_{i-1}, r_i) = \text{Pairs}(W)$. Due to property (2) we now also know $\bigcup_{r \in I} \text{pairsplit}(W, \text{Pairs}(W), r) = \text{Pairs}(W)$. □

5.2.2 Calculating the PairLB

To actually calculate a lower bound for this instance of Set Cover we could use the same method we used for the ImpLB. However our preliminary tests showed that this method does not yield very good results if we use it to calculate the PairLB. This is probably due to the fact that we use RefAll(W) instead of Ref(W).

Instead we translate the problem instance into an ILP instance. We can then create an LP relaxation of that instance and solve this relaxation with an LP solver. Clearly the solution for the LP relaxation is a lower bound for the solution of the ILP instance since we are dealing with a minimization problem.

To construct the ILP instance we will use the standard ILP formulation for Set Cover. First, we introduce a binary variable x_r for every one-step-refinement $r \in \text{RefAll}(W)$. We want x_r to be equal to 1 if and only if the pairsplit set of r is part of the set cover solution. Next we are going to introduce a constraint $1 \leq \sum_{r \in S_p} x_r$ for every pair $p \in \text{Pairs}(W)$ where $S_p := \{r \in \text{RefAll}(W) \mid p \in \text{pairsplit}(W, \text{Pairs}(W), r)\}$ is the set of all one-step-refinements that split p. The objective function is then just to minimize the sum of all variables. The LP relaxation is obtained by allowing the variables to have any value between 0 and 1. This of course also makes it possible that the result of the objective function will not be an integer. In that case we will simply round up to the nearest integer.

To reduce the size of the LP instance and speed up the calculation we can again use one of the data reduction techniques that we used for the ImpLB. This time we can eliminate all subsets S that come from one-step-refinements that are not applied to the root of the tree. Since we are using one-step-refinements from RefAll(W) we can always apply the same one-step-refinement at the root of W and get a subset that is a superset of S.

5.3 Evaluation

Next, we will evaluate the effect that the two lower bounds have on the running time of the algorithm. As mentioned above, we calculate the lower bounds after Line 5 in each call of Refine in Algorithm 1 and return $\perp$ if one of them is bigger than the remaining size budget. However, preliminary experiments showed that the calculation of the PairLB takes too long compared to how effective it is. Because of this we will only calculate the PairLB once to obtain an initial lower bound for the solution of the MDTS instance as mentioned in Section 3.2.

Now we compare a version of our algorithm that uses all improvements we have shown so far including the lower bounds with a version that uses all improvements except the lower bounds. We will refer to these algorithms as *Imp3* and *Imp2*, respectively.

Figure 5 shows this comparison. Out of the 700 total instances Imp2 was able to solve 260 instances. Imp3 was able to solve the same 260 instances plus an additional 60 instances. Also, all of those 260 instances were solved faster by Imp3 than they were solved by Imp2. Imp3 also decreases the size of the search tree drastically. On average, the search trees of Imp3 had 85% less nodes than the search trees of Imp2 on the same

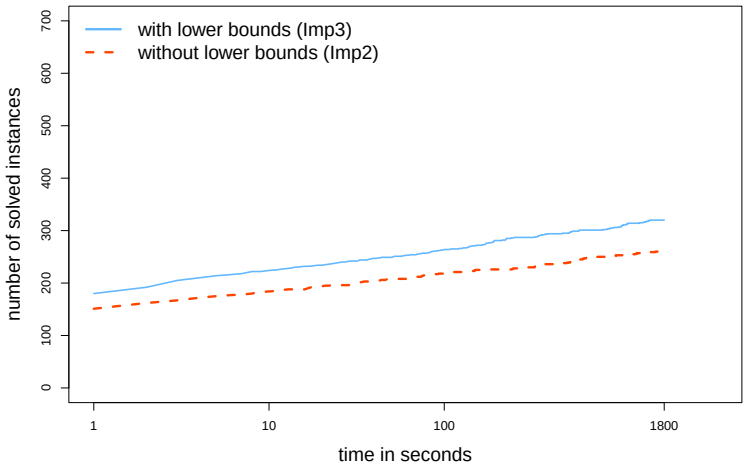


Figure 5: Comparison of the algorithm with and without lower bounds.

instances. All of this shows that the two lower bounds are a big improvement for the algorithm in every instance.

6 Subset Constraints

In this section we will introduce subset constraints. A subset constraint of a vertex v in a witness tree W is a subset $S \subseteq E[W, v]$ which imposes the constraint that one-step-refinements are not allowed to remove all examples in S from the subtree of v. The idea is that if such one-step-refinements lead to a correct tree then there would be a different correct tree that does not violate this constraint. We formalize the idea of a subset constraint in the following definition.

Definition 6.1. Let W be a witness tree and let $v \in V(W)$ be some vertex in 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 $\text{Const}(W, v)$ contains all subset constraint of v in the tree W.

The intuition behind subset constraints is that sometimes a witness tree W can be transformed into a slightly different witness tree W' without increasing the size and without changing the classes that the tree assigns to the examples. This could for example be done by simply changing the threshold t of a vertex v in W to a different threshold t'. This does not change the size of W and as long as there are no examples in the subtree of v that have values between t and t' then there is no example that gets assigned to a different leaf by this change. That means the algorithm can skip the search tree node N with $\text{Tree}(N) = W$. If one of these search tree nodes leads to a correct tree then there would be a different search tree node that leads to the same tree where t is replaced by t'.

However, detecting that there are no examples inbetween t and t' by simply checking every pair of thresholds for every vertex is inefficient. Instead we want to use a subset constraint C to keep track of the values inbetween t and t'. We would add C when v is first added to the tree with the threshold t. Then, every time a one-step-refinement changes the tree we update which examples of C are still in the subtree of v. That way we immediatly know when C is violated.

To prove the correctness of the specific subset constraints we will introduce later in this section, we first need to introduce the concept of a refinement.

6.1 Refinements

We want to be able to show that a witness tree R was created by applying one or more one-step-refinements to a witness tree W by looking at the structure of both trees. For this we will look at the following properties of one-step-refinements.

First we can notice that a one-step-refinement in $\text{Ref}(W)$ never changes any vertices that already exist in W, it only adds an inner vertex and a leaf. The role of a vertex also never changes: a leaf will always be a leaf and an inner vertex will always be an inner vertex. The relative positions of vertices to eachother do not change either. This means that if a vertex is in the left subtree of some vertex then it will always be in that left subtree. The same is true for the vertices in the right subtree. Lastly we know that a witness will always stay in its leaf. This leads to the following definition.

Definition 6.2. Let $W = (T, \text{cut}, \text{cla}, \text{wit})$ and $R = (T', \text{cut}', \text{cla}', \text{wit}')$ 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 it is a leaf in R.
3. Let $v_1, v_2 \in V(W)$ be a pair of vertices. The vertex v_1 is in the left subtree of v_2 in W if and only if it is in the left subtree of v_2 in R. The vertex v_1 is in the right subtree of v_2 in W if and only if it is in the right subtree of v_2 in R.
4. For every leaf $\ell \in V(W)$ we have $\text{cla}(\ell) = \text{cla}'(\ell)$ and $\text{wit}(\ell) \in E[R, \ell]$.
5. For every inner vertex $v \in V(W)$ we have $\text{cut}(v) = \text{cut}'(v)$.

Clearly, if R was created by the algorithm by applying one or multiple one-step-refinements to W then R is a refinement of W. Now we also want to show the other direction. For this we use the following lemma.

Lemma 6.1. *Let R be a correct witness tree that is a refinement of the witness tree W and let $e \in \text{Dirty}(W)$ be a dirty example in W. Then, there must be a one-step-refinement $r \in \text{Ref}(W)$ with $W \xrightarrow{r} W'$ such that R is a refinement of W'.*

Proof. We need to find a one-step-refinement $r = (v, i, t, e)$ with $W \xrightarrow{r} W'$ such that R is a refinement of W'. That means the leaf ℓ and the inner vertex u we add with r need to be in $V(R)$ due to Property 1 in Definition 6.2. To identify these vertices we can use the dirty example e. Let ℓ' be the leaf that e is assigned to in W. The one-step-refinement r will make e the witness of ℓ. Property 4 tells us that ℓ must be the leaf that e is assigned to in R. The inner vertex u must now be the vertex in R that has ℓ in one of its subtrees and ℓ' in the other subtree. There can only be one vertex like this. This inner vertex u now tells us the cut (i, t) we need for r due to Property 5. Now we just need to find the vertex v in $V(W)$ at which r will be applied. For this we can look at the path from u to ℓ' in R. Starting from u, the vertex v must be the first vertex on that path that also exists in $V(W)$. □

An important detail of this lemma is that W needs to have at least one dirty example. This means we may not be able to create R by applying one-step-refinements to W if there is always a smaller correct tree of which R is also a refinement. However, this is sufficient for our purposes since our goal is to find the smallest correct tree.

6.2 Definition and Proof of Subset Constraints

Now we are going to properly define and prove the subset constraints that we are going to use to improve the algorithm. In order to show their correctness we are first going to introduce Definition 6.3. This definition assumes that subset constraints can be added to a tree after a one-step-refinement has been applied. We will define the exact conditions for this later in this section when we introduce the specific subset constraints.

Definition 6.3. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \text{Tree}(N)$ and the dirty example $e := \text{ex}(N)$ and let $r := (v, i, t, e) \in \text{Ref}(W)$ 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 correct 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 also correct.
2. R' is not bigger than R.
3. There is a different one-step-refinement $r' := (v', i', t', e) \in \text{Ref}(W)$ with $W \xrightarrow{r'} W''$ that Algorithm 1 chooses before r such that R' is a refinement of W''.

The most important part of Definition 6.3 is Property 3. It requires that Algorithm 1 chooses r' before r. Together with the other properties this means, according to Lemma 6.1, there must be a correct tree R'' that is not bigger than R, of which R' is a refinement and which is discovered by the algorithm before R.

That means, if there is a correct tree W that violates a correct subset constraint there must be a different correct tree W' that is discovered by the algorithm before W and is not bigger than W. If W' now also violates a correct subset constraint we can find a correct tree W'' that is discovered by the algorithm before W' and is not bigger than W'. We can keep doing this until we eventually find a correct tree that does not violate any correct subset constraints. Due to Definition 6.3 and there only being a finite number of trees, such a tree must always exist. We formalize this idea with the following theorem.

Theorem 6.2. *Let $s \in \mathbb{N}$. If there is at least one correct witness tree of size at most s, then there is at least one correct witness tree of size at most s that does not violate any correct subset constraints.*

Proof. Let us assume there is a correct witness tree W that has at most size s and violates at least one correct subset constraint C. According to Definition 6.3 there must be a different correct witness tree W' that is not bigger than W and is discovered by the algorithm before W.

We can keep replacing the current tree with a different tree according to Definition 6.3 until we find a tree where no correct subset constraint is violated. Since we know that the replacement tree is always discovered earlier by the algorithm and there is only a finite number of trees we must find such a tree eventually. □

With this we will now introduce two specific subset constraints. To show that the algorithm can skip any tree where one of these subset constraints is violated we just need to show that they are correct.

6.3 Threshold Subset Constraints

First we introduce a subset constraint that uses the idea from the example at the beginning of this section where we showed that we can sometimes replace a threshold t with a different threshold t' without changing the leaf that any example is assigned to.

Definition 6.4. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \text{Tree}(N)$ and the dirty example $e := \text{ex}(N)$ and let $(v, i, t, e), (v, i, t', e) \in \text{Ref}(W)$ be two one-step-refinements with $W \xrightarrow{(v,i,t,e)} R$ and $W \xrightarrow{(v,i,t',e)} R'$ such that we have $e[i] \leq t' < t$ or $t < t' < e[i]$. Finally, let ℓ and u be the leaf and inner vertex that are added to W by (v, i, t, e).

Then, we add the *Threshold Subset Constraint* $C := E[R, \ell] \setminus E[R', \ell]$ to $\text{Const}(R, u)$ and we call t' the *Constraint Threshold* of C.

Clearly if a Threshold Subset Constraint of a vertex u in a witness tree W is violated then replacing the threshold of u by the Constraint Threshold t' will not change the leaf of any example in W. We will now use this observation to show that a Threshold Subset Constraint is always correct.

Theorem 6.3. *Threshold Subset Constraints are correct.*

Proof. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \text{Tree}(N)$ and the dirty example $e := \text{ex}(N)$ and let $(v, i, t, e), (v, i, t', e) \in \text{Ref}(W)$ be two one-step-refinements with $W \xrightarrow{(v,i,t,e)} R$ and $W \xrightarrow{(v,i,t',e)} R'$ such that we have $e[i] \leq t' < t$ or $t < t' < e[i]$. Finally, let ℓ and u be the leaf and inner vertex that are added to W by (v, i, t, e) and let $C \in \text{Const}(R, u)$ be the Threshold Subset Constraint added to u in R.

Next, let R^* be a correct witness tree that is a refinement of R and violates C. We find a different witness tree that has the three properties from Definition 6.3. We can create such a witness tree R^{**} by simply replacing the threshold of u in R^* by the Constraint Threshold t'. Since C is violated in R^* we know that R^{**} is a correct tree. We also have not increased the size of the tree. Now we just need to show Property 3.

In Section 3 we specified that the algorithm chooses (v, i, t', e) before (v, i, t, e) because t' is closer to $e[i]$ than t. The only difference between R and R' and between R^* and R^{**} is the threshold of u. That means R^{**} is a refinement of R'. □

6.4 Dirty Subset Constraints

The second subset constraint we want to introduce is the Dirty Subset Constraint. It is based on the idea that we do not always have to apply a one-step-refinement at every vertex on the leaf to root path. If for example we want to apply a one-step-refinement at a vertex v but one of the child subtrees of v is already correct, then we could just apply the same one-step-refinement at the other child subtree. Since there are no dirty examples in the correct subtree it makes no sense to apply the one-step-refinement right above that subtree. This leads to the following definition.

Definition 6.5. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \text{Tree}(N)$ and the dirty example $e := \text{ex}(N)$, let v be an inner vertex in W with the children v_1 and v_2 and let $(v, i, t, e), (v_1, i, t, e) \in \text{Ref}(W)$ be two one-step-refinements with $W \xrightarrow{(v,i,t,e)} R$ and $W \xrightarrow{(v_1,i,t,e)} R_1$. Finally, let ℓ and u be the leaf and inner vertex that are added to W by (v, i, t, e).

Then, we add the *Dirty Subset Constraint* $C = E[W, v_2] \cap \text{Dirty}(W)$ to $\text{Const}(R, u)$.

Theorem 6.4. *Dirty Subset Constraints are correct.*

Proof. Let N be a node in the search tree of Algorithm 1 with the witness tree $W := \text{Tree}(N)$ and the dirty example $e := \text{ex}(N)$, let v be an inner vertex in W with the children v_1 and v_2 and let $(v, i, t, e), (v_1, i, t, e) \in \text{Ref}(W)$ be two one-step-refinements with $W \xrightarrow{(v,i,t,e)} R$ and $W \xrightarrow{(v_1,i,t,e)} R_1$. Finally, let ℓ and u be the leaf and inner vertex that are added to W by (v, i, t, e).

Next, let R^* be a correct witness tree that is a refinement of R and violates C. We find a different witness tree R^{**} that has the three properties from Definition 6.3. Without loss of generality, we assume that v_1 is the left child of v. We now obtain R^{**} in the following way.

First we know that v and u still exist in R^* but u may no longer be the direct parent of v. So we are now going to take the entire subtree of u with the exception of the subtree of v, remove the edge from v to its left child and move the subtree of u into that gap. We connect everything so that the previous parent of u is now the parent of v and the previous parent of v is now the parent of the previous left child of v. Lastly we replace the right subtree of v in R^{**} with the right subtree of v in W.

We know that $E[R^{**}, u] \subseteq E[R^*, u]$ which means that everything in the subtree of u is still correctly classified. We also know that $E[R^*, v] \subseteq E[R^{**}, v]$. But we can also see that $E[R^{**}, v] \subseteq E[W, v]$. We replaced the right subtree of v in R^{**} with the right subtree of v in W and we know that C is violated which means all dirty examples in that right subtree were removed. This means that R^{**} is correct.

These changes also do not increase the size of the tree and they also do not change a leaf into an inner vertex or an inner vertex into a leaf. If we now only look at the vertices that were already present in R we can notice that the relative position of most of these vertices has not changed. The only changes are that u and ℓ are now in the left subtree of v instead of v being a child of u. This is the exact same difference that we have between R and R_1. That means R^{**} is a refinement of R_1.

Since the algorithm chooses (v_1, i, t, e) before (v, i, t, e) we have shown that R^{**} fulfills the three properties from Definition 6.3. □

6.5 Evaluation

Now we will evaluate the effect that the two subset constraints have on the running time of the algorithm. For this we will check wether any subset constraint is violated right after a one-step-refinement has been performed in Line 9 of Algorithm 1.

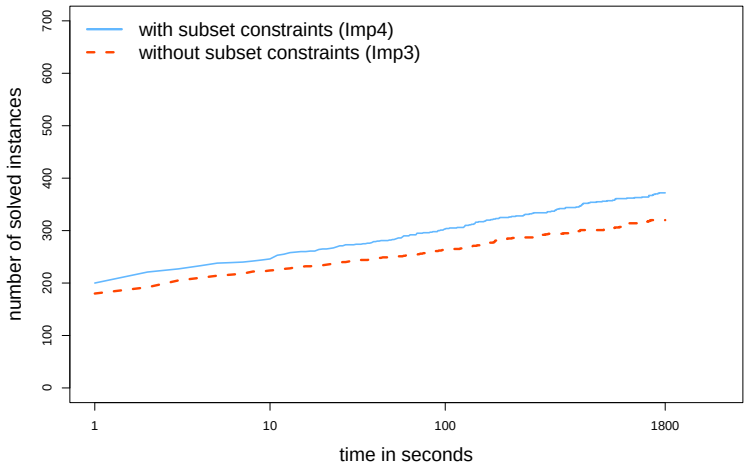


Figure 6: Comparison of the algorithm with and without subset constraints.

Now we will compare a version of our algorithm that uses all improvements we have shown so far including the subset constraints with a version that uses all improvements except the subset constraints. We will refer to these algorithms as *Imp4* and *Imp3*, respectively.

Figure 6 shows this comparison. In total, Imp3 solved 320 out of the 700 instances while Imp4 was able to solve 372 instances. Additionally, there was only one instance that was solved by Imp3 but not by Imp4. Out of all instances that were solved by both algorithms, there were only 13 where Imp4 took more than one second longer than Imp3. The size of the search trees also decreased significantly due to the subset constraints. On average, the search trees of Imp4 had 38% less nodes than the search trees of Imp3 on the same instances. This shows that the two subset constraints are also generally a big improvement for the algorithm.

7 Subset Caching

The improvement in this section is inspired by the caching of subproblems used by Demirovic et al. [10]. With their MurTree algorithm, they solve the problem of minimizing the amount of dirty examples of a witness tree under a size and depth constraint. Their algorithm works by first iterating through all possible cuts for the root of the tree and then recursively calculating the optimal trees for the left and right child for all possible ways of splitting up the remaining size budget between the two sides. This approach naturally lends itself to saving and then later reusing the solutions to these subproblems. Our algorithm, however, does not naturally calculate solutions for subproblems. This is because our algorithm can modify almost any part of the tree at any time which means the algorithm never focuses on correctly classifying just a specific subset of the examples. Because of that, we modify the subproblem caching approach in the following way.

We will use a data structure to save lower bounds for specific subsets of the examples. These lower bounds tell us how many inner vertices we would need at least to correctly classify that subset of the examples. In any search tree node N with $W := \text{Tree}(N)$, we can 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 of L is present in our data structure. We then know that the lower bound associated with that subset is also a lower bound for L. Since all examples in L are assigned to the same leaf, that lower bound must also be a lower bound for the whole tree W. And if that lower bound is bigger than the remaining size budget s', we know that it is not possible to correctly classify the dataset by applying at most s' one-step-refinements to W. Consequently, the algorithm can return $\perp$. If the lower bound 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.

Of course for all of this to work, we need a data structure that lets us save lower bounds for subsets of the examples and lets us quickly check out all subsets of a specific set of examples. For this, we use the set-trie data structure proposed by Savnik [15]. A set-trie saves the subsets in a tree structure where each vertex represents a single example. We add a subset of examples to the set-trie by first sorting them in ascending order. We then start at the first example in the order and check if the root of the set-trie has a child c that represents that example. If not, then we add such a child. We then check if c has a child that represents the second example and add a new child if that is not the case. We continue doing this until we have created a path that represents the entire subset. We then mark the last vertex in this path as an end vertex and save the lower bound of the subset in this end vertex.

The last step is to find a way to actually populate the set-trie with lower bounds for relevant subsets of the examples. As mentioned above our algorithm does not naturally calculate lower bounds for subsets of examples. Instead, we do the following. In a search tree node N, we look at the example set $L := E[\text{Tree}(N), \ell]$ of any leaf ℓ with $|L| \leq 30$. Let r be the remaining size budget of the current tree. We then run a separate instance of our algorithm that checks whether the set L can be correctly classified with a tree

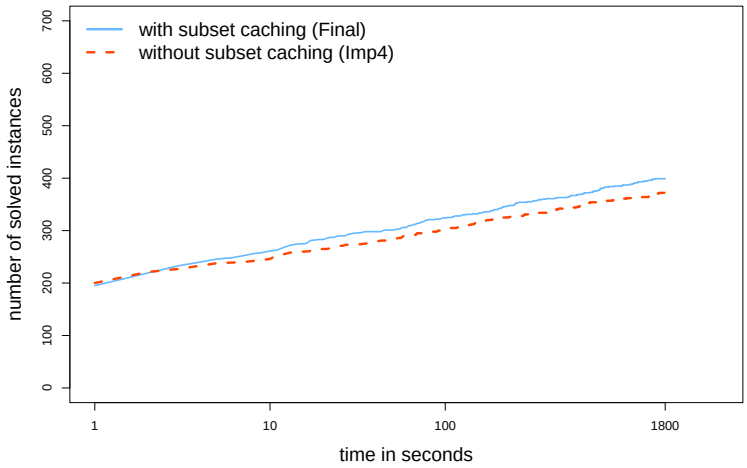


Figure 7: Comparison of the algorithm with and without subset caching.

of at most size r. If that is not the case, then we can add L to the set-trie with the lower bound $r + 1$. Of course we could then check if L can be correctly classified with a tree of size at most $r + 1$ to improve the lower bound, but we decided to only check size r 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. The reason why we chose to limit the size of L to at most 30 is because doing this without such a limit would take too much time and preliminary experiments show that 30 is a good limit.

Lastly, we want to mention that the set-trie is especially useful when solving MDTS instead of only DTS since we do not have to throw the set-trie away after the algorithm has checked a single s. We can keep reusing the set-trie until we find the optimal s.

7.1 Evaluation

We will now compare a version of our algorithm that uses all improvements we have shown so far including subset caching with a version that uses all improvements except subset caching. We will refer to these algorithms as *Final* and *Imp4*, respectively. Imp4 solved 372 out of the 700 total instances. Final was able to solve 399 instances and there were only 4 instances that were solved by Imp4 but not by Final.

This shows that subset caching is an improvement for the algorithm. However, on average, the search trees of Final had 580% more nodes than the search trees of Imp4 on the same instances. This is clearly due to the additional instances of DTS that we solve to populate the set-trie and it shows why it is important for us to limit which example subsets we choose to check and potentially add to the set-trie.

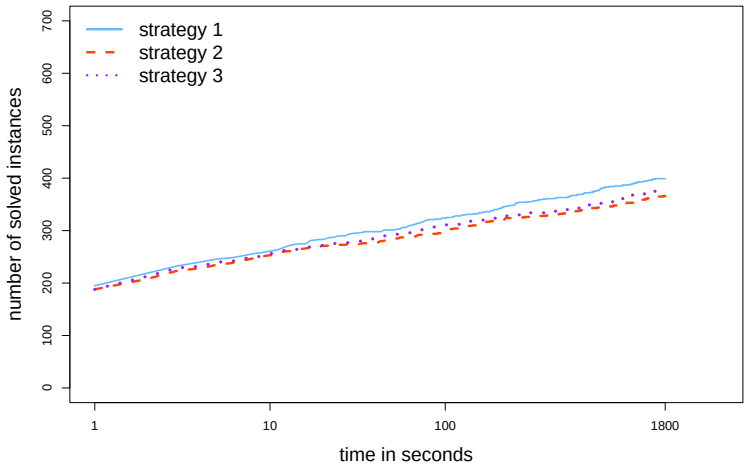


Figure 8: Comparison of the three strategies for solving MDTS.

8 Final Evaluation

In this section, we evaluate our algorithm with all improvements that we presented in the previous sections. We will start by introducing two more strategies for solving MDTS using Algorithm 1. We then compare the two strategies with our original strategy from Section 3.2. Finally, we will compare the best strategy with the two SAT-based algorithms from Narodytska et al. [3] and Janota et al. [4].

8.1 Comparison of the three Strategies

Our algorithm only solves the decision problem DTS. But since we are insterested in the optimization problem MDTS we presented a strategy that allows us to solve MDTS by solving multiple instances of DTS using our algorithm.

Our original strategy from Section 3.2 linearly increases the maximum size of the tree starting from a lower bound until our algorithm no longer returns $\perp$. However, this is not the only way to find the optimal value of s.

As a second strategy we can linearly decrease the maximum size of the tree starting from an upper bound until our algorithm returns $\perp$. And as a third strategy we can perform a binary search between a lower bound and an upper bound until the optimal tree size is found.

For the initial lower bound we use the PairLB from Section 5.2. For the initial upper bound we will use the scikit-learn [16] implementation of the CART algorithm originally proposed by Breiman et al. [6]. Janota et al. [4] also use this implementation to calculate an upper bound.

Figure 8 shows the running time comparison of all three strategies. Generally all three strategies perform roughly the same, with Strategy 1 being slightly better than the others in the long run. Strategy 1 solved 399 out of the 700 instances. Strategies 2

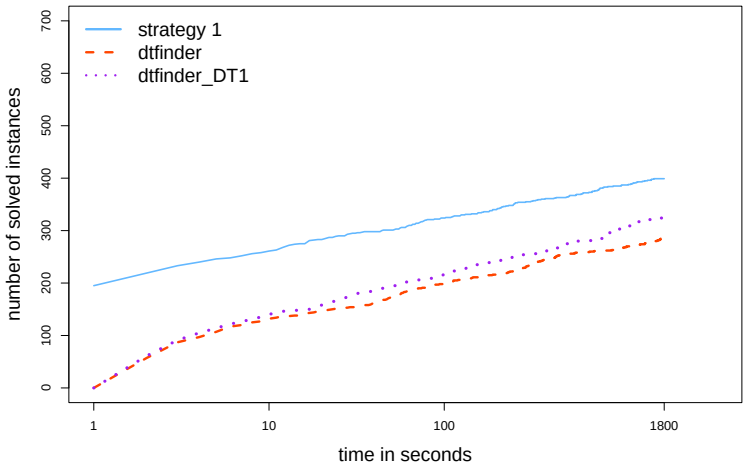


Figure 9: Comparison of strategy 1 with the two SAT-based algorithms.

and 3 solved 366 and 377 instances, respectively. The 399 instances solved by Strategy 1 include all except one of the instances that can be solved by at least one of the other two strategies.

The reason why the three strategies perform similarly is most likely because there are always at least two instances of DTS that need to be solved by all strategies. If s_{min} is the solution for an instance of MDTS, then all strategies need to at least solve the DTS instance with $s = s_{min}$ and the instance with $s = s_{min} - 1$. Since these two instances are closest to the optimal value s_{min} they take the longest time to solve and are therefore the main deciding factor in wether a strategy can solve an instance of MDTS or not.

If we compare the running time of the three strategies on instances that can be solved by all of them we can see that Strategy 1 is only very rarely substantially slower than the other two: There are only 16 instances where Strategy 1 is more than 10 seconds slower than the fastest strategy. This shows that Strategy 1 is generally the best choice for our algorithm.

8.2 Comparison of our Algorithm and the SAT Algorithms

Now we will compare Strategy 1 with the SAT-based algorithms by Narodytska et al. [3] and Janota et al. [4]. We will use the names *dtfinder_DT1* and *dtfinder*, respectively, to refer to these algorithms. As mentioned in Section 1, we will use the improved version of the encoding by Narodytska et al. [3] that was presented by Janota et al. [4].

Figure 9 shows the running time comparison between Strategy 1, dtfinder_DT1 and dtfinder. Overall, Strategy 1 is much better than the other two algorithms. Strategy 1 can solve 399 instances in total compared to the 286 and 324 total instances solved by dtfinder and dtfinder_DT1, respectively. There are also 324 instances that Strategy 1 can solve in less than 100 seconds each. Additionally, there are only three instances that can be solved by dtfinder or dtfinder_DT1 but not by Strategy 1.

Again we can compare the running time of all three algorithms on instances that can be solved by all of them. This time there are only 9 instances where Strategy 1 is slower than the fastest algorithm. This also shows that Strategy 1 is the best choice on almost all instances.

9 Conclusion

As our experiments showed, our algorithm was able to solve roughly 25% more instances than the state of the art algorithms by Narodytska et al. [3] and Janota et al. [4]. The improvements that played the biggest role in achieving this result were the dirty example priority from Section 3.1, the lower bounds from Section 5, the subset constraints from Section 6, and the subset caching from Section 7. The reduction rules from Section 4 only slightly improved the running time of the algorithm. However, these reduction rules are a good protection against data with a lot of redundancy.

We also showed that the strategy for solving MDTS using multiple instances of DTS does not have a big impact on the overall running time. We concluded that this is because the DTS instances that need to be solved by all strategies are the ones that take the longest time to solve.

Another improvement that did not help as much as we had hoped is the PairLB from Section 5.2. We already mentioned that we only used this lower bound as an initial lower bound for s when solving MDTS. This is because calculating this lower bound in every node of the search tree did not lead to an improvement in the overall running time. However, we did observe that the number of search tree nodes still decreased by up to two orders of magnitude when we used the PairLB in every node. This suggests that a potential faster way of calculating the PairLB would lead to a large improvement in the overall running time.

Another avenue for future research could be to adapt our algorithm and the individual improvements to datasets with more than two classes. We assume that with few modifications most of the improvements can naturally support more than two classes.

Our algorithm could also be adapted to other problems. We already mentioned in Section 1 that some algorithms try to optimize the depth of the tree instead of the size. Again, we assume that most of the improvements should naturally work with a depth constraint instead of a size constraint.

Another problem that many algorithms focus on is to minimize the number of dirty examples while the tree has a size or depth constraint. We assume that it would be a lot more difficult to adapt our algorithm and the improvements to these problems since many of the improvements use the assumption that we want to correctly classify all examples.

Finally, as we have mentioned multiple times already, our base algorithm is a special case of the witness tree algorithm by Komusiewicz et al. [1]. Clearly, another natural avenue for future research would be to adapt our improvements to the tree ensemble version of MDTS.

References

[1] C. Komusiewicz, P. Kunz, F. Sommer, and M. Sorge, "On computing optimal tree ensembles," in *Proceedings of the International Conference on Machine Learning, ICML '23*, vol. 202 of *Proceedings of Machine Learning Research*, pp. 17364–17374, PMLR, 2023.

[2] J. D. Romano, T. T. Le, W. La Cava, J. T. Gregg, D. J. Goldberg, P. Chakraborty, N. L. Ray, D. Himmelstein, W. Fu, and J. H. Moore, "Pmlb v1.0: an open source dataset collection for benchmarking machine learning methods," *arXiv preprint arXiv:2012.00058v2*, 2021.

[3] N. Narodytska, A. Ignatiev, F. Pereira, and J. Marques-Silva, "Learning optimal decision trees with SAT," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI '18* (J. Lang, ed.), pp. 1362–1368, ijcai.org, 2018.

[4] M. Janota and A. Morgado, "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* (L. Pulina and M. Seidl, eds.), vol. 12178, pp. 501–518, Springer, 2020.

[5] J. N. Morgan and J. A. Sonquist, "Problems in the analysis of survey data, and a proposal," *Journal of the American Statistical Association*, vol. 58, no. 302, pp. 415–434, 1963.

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

[7] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is np-complete," *Inf. Process. Lett.*, vol. 5, no. 1, pp. 15–17, 1976.

[8] V. G. Costa and C. E. Pedreira, "Recent advances in decision trees: an updated survey," *Artif. Intell. Rev.*, vol. 56, no. 5, pp. 4765–4800, 2023.

[9] S. Verwer and Y. Zhang, "Learning optimal classification trees using a binary linear program formulation," in *Proceedings of the 31st Conference on Artificial Intelligence, AAAI '19*, pp. 1625–1632, AAAI Press, 2019.

[10] E. Demirovic, A. Lukina, E. Hebrard, J. Chan, J. Bailey, C. Leckie, K. Ramamohanarao, and P. J. Stuckey, "Murtree: Optimal decision trees via dynamic programming and search," *J. Mach. Learn. Res.*, vol. 23, pp. 26:1–26:47, 2022.

[11] D. Bertsimas and J. Dunn, "Optimal classification trees," *Mach. Learn.*, vol. 106, no. 7, pp. 1039–1082, 2017.

[12] J. Lin, C. Zhong, D. Hu, C. Rudin, and M. I. Seltzer, "Generalized and scalable optimal sparse decision trees," in *Proceedings of the 37th International Conference*

on Machine Learning, ICML '20, vol. 119 of *Proceedings of Machine Learning Research*, pp. 6150–6160, PMLR, 2020.

[13] F. Avellaneda, "Efficient inference of optimal decision trees," in *Proceedings of the 34th Conference on Artificial Intelligence, AAAI '20*, pp. 3195–3202, AAAI Press, 2020.

[14] A. Schidler and S. Szeider, "Sat-based decision tree learning for large data sets," in *Proceedings of the 35th Conference on Artificial Intelligence, AAAI '21*, pp. 3904–3912, AAAI Press, 2021.

[15] I. Savnik, "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* (A. Cuzzocrea, C. Kittl, D. E. Simos, E. R. Weippl, and L. Xu, eds.), vol. 8127 of *Lecture Notes in Computer Science*, pp. 134–148, Springer, 2013.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

Selbstständigkeitserklärung

Hiermit versichere ich, Luca Pascal Staus, dass ich die vorliegende Arbeit selbstständig verfasst, ganz oder in Teilen noch nicht als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet. Mir ist bewusst, dass es sich bei Plagiarismus um akademisches Fehlverhalten handelt, das sanktioniert werden kann.

Ort, Datum Unterschrift