Learning Small Decision Trees with Large Domain (Supplementary Material: Full Paper)

Eduard Eiben[1] , **Sebastian Ordyniak**[2] and **Giacomo Paesani**[2] and **Stefan Szeider**[3]
[1]Royal Holloway University of London, UK
[2]University of Leeds, UK
[3]Algorithms and Complexity Group, TU Wien, Vienna, Austria
eduard.eiben@rhul.ac.uk, {s.ordyniak, g.paesani}@leeds.ac.uk, sz@ac.tuwien.ac.at

Abstract

One favors decision trees (DTs) of the smallest size or depth to facilitate explainability and interpretability. However, learning such an optimal DT from data is well-known to be NP-hard. To overcome this complexity barrier, Ordyniak and Szeider (AAAI 21) initiated the study of optimal DT learning under the parameterized complexity perspective. They showed that solution size (i.e., number of nodes or depth of the DT) is insufficient to obtain fixed-parameter tractability (FPT). Therefore, they proposed an FPT algorithm that utilizes two auxiliary parameters: the maximum difference (as a structural property of the data set) and maximum domain size. They left it as an open question of whether bounding the maximum domain size is necessary.

The main result of this paper answers this question. We present FPT algorithms for learning a smallest or lowest-depth DT from data, with the only parameters solution size and maximum difference. Thus, our algorithm is significantly more potent than the one by Szeider and Ordyniak as it can handle problem inputs with features that range over unbounded domains. We also close several gaps concerning the quality of approximation one obtains by only considering DTs based on minimum support sets.

1 Introduction

Decision Trees (DTs) have proved to be extremely useful tools for describing, classifying, and generalizing data [Larose and Larose, 2014; Murthy, 1998; Quinlan, 1986]. Because of their simplicity, DTs are particularly attractive for providing interpretable models of the underlying data, an aspect whose importance has been strongly emphasized over recent years [Darwiche and Hirth, 2023; Doshi-Velez and Kim, 2017; Goodman and Flaxman, 2017; Lipton, 2018; Monroe, 2018]. In this context, one prefers *small trees* (trees of small size or small depth), as they are easier to interpret and require fewer tests to make a classification. Small trees are also preferred in view of the parsimony principle (Occam's Razor) since small trees are expected to generalize better to new data [Bessiere *et al.*, 2009]. However, learning small trees is computationally costly: it is NP-hard to decide whether a given data set can be represented by a DT of a certain size or depth [Hyafil and Rivest, 1976]. In view of this complexity barrier, several methods based on branch & bound algorithms, constraint programming, mixed-inter programming, or satisfiability solving have been proposed for learning small DTs [Avellaneda, 2020; Bessiere *et al.*, 2009; Aglin *et al.*, 2020a; Aglin *et al.*, 2020b; Bertsimas and Dunn, 2017; Demirovic *et al.*, 2022; Hu *et al.*, 2020; Janota and Morgado, 2020; Narodytska *et al.*, 2018; Shati *et al.*, 2021; Schidler and Szeider, 2021; Verhaeghe *et al.*, 2020; Verwer and Zhang, 2017; Verwer and Zhang, 2019; Zhu *et al.*, 2020]. This bulk of recent empirical work underlines the importance of computing optimal decision trees.

In this paper, we investigate the problem of finding small decision trees (w.r.t. size or depth) under the framework of *Parameterized Complexity* [Downey and Fellows, 2013; Gottlob *et al.*, 2002; Niedermeier, 2006]. This framework allows us to achieve a more fine-grained and qualitative analysis, revealing properties of the input data in terms of *problem parameters* that provide run-time guarantees for decision tree learning algorithms. The key notion of Parameterized Complexity is *fixed-parameter tractability (*FPT*)* which generalizes the classical polynomial time tractability by allowing the running time to be exponential in a function of the problem parameters while remaining polynomial in the input size (we provide more detailed definitions in Section 2). Fixed-parameter tractability captures the scalability of algorithms to large inputs as long as the problem parameters remain small. Several fundamental problems that arise in AI have been studied in terms of their fixed-parameter tractability, including Planning [Bäckström *et al.*, 2012], SAT and CSP [Bessière *et al.*, 2008; Gaspers *et al.*, 2017], Computational Social Choice [Bredereck *et al.*, 2017], Machine Learning [Ganian *et al.*, 2018], and Argumentation [Dvorák *et al.*, 2012].

For DT learning, we consider parameterizations of the following two fundamental NP-hard problems:

MINIMUM DECISION TREE SIZE (DTS): we are given a set of examples, labelled positive or negative, each over a set of features; each feature f ranges over a linearly ordered range of possible values (by choosing an arbitary ordering this also captures categorical data), and an integer s (for *size*). The task is to find a DT of minimum size or report correctly that no decision tree with at most s nodes exists. Here we

parameters		complexity
solution size	maximum difference	FPT †
solution size	-	W[2]-hard‡, in XP‡
-	maximum difference	para-NP-hard‡
-	-	para-NP-hard‡

Table 1: † this paper, Theorem 4; ‡ are results by Ordyniak and Szeider [2021].

consider DTs where each node tests whether a certain feature is below a certain threshold or not.

MINIMUM DECISION TREE DEPTH (DTD) is defined similarly, where instead of the bound s on the total number of nodes, a bound d (for *depth*) on the number of nodes on any root-to-leaf path is provided.

For both problems, it is natural to include the *solution size* (i.e., s for DTS and d for DTD, respectively) as a parameter since our objective is to learn DTs where these values are small. However, Ordyniak and Szeider's [2021] complexity analysis revealed that solution size is not sufficient to obtain fixed-parameter tractability. They, therefore, proposed two additional problem parameters: (i) the *maximum domain size*, i.e., the largest number of different values a feature ranges over, and (ii) the *maximum difference*, i.e., the largest number of features two examples with a different classification can disagree in. With these two additional parameters at hand, Ordyniak and Szeider could show that DTS and DTD are fixed-parameter tractable. They showed that without including the maximum difference in the parameterization, one loses fixed-parameter tractability. However, they left it open whether the maximum domain size is indeed needed as a parameter.

In this paper, we answer this open problem, obtaining fixed-parameter tractability of DTS and DTD just with the two parameters solution size and maximum difference. Our main result can be stated as follows.

- DTS *and* DTD *are fixed-parameter tractable parameterized by solution size and maximum difference (Theorem 4).*

This result completes Ordyniak and Szeider's parameterized complexity classification, as shown in Table 1.

Our result is surprising, as for similar problems, the domain size must be included in the parameterization. For instance, the Constraint Satisfaction Problem (CSP) is fixed-parameter tractable by the combined parameter primal treewidth and domain size [Gottlob *et al.*, 2002; Samer and Szeider, 2010], and by the combined parameter strong backdoor size and domain size [Gaspers *et al.*, 2017]; in both cases the problem becomes W[1]-hard (and hence fixed-parameter *in*tractable) when domain size is dropped from the parameterization.

Our result has a beneficial algorithmic impact. As we do not need to parameterize by maximum domain size, we have a significantly more powerful algorithm that allows us to compute optimal DTs (in terms of smallest depth/size) even for instances where features range over a large set of possible values. What makes our result further appealing is that the maximum difference, the only additional parameter we need in addition to solution size, is quite small in real-world data sets. Ordyniak and Szeider [2021] list values for various standard benchmark sets from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml). In some cases, the maximum difference is two orders of magnitude smaller than the number of examples or features.

A subset of the features that suffices to correctly classify a classification instance is called a *support set*. Ordyniak and Szeider [2021] observed that, in general, a small or low-depth DT would not necessarily use a smallest support set. Indeed, this property of small or low-depth DTs provides a challenge to algorithmically finding such DTs, as we cannot first minimize the feature set in a preprocessing phase if we want to find DTs of the smallest size or lowest depth. In the second part of this paper, we quantify the impact on the size and depth of DTs when minimizing first the feature set. It turns out that regarding this question, it is significant whether the considered data is over features with unbounded domain size or if the domain size is bounded. For the unbounded domain case we obtain the following result.

- *The smallest size (depth) of a DT for a classification instance (with unbounded domain) using only features of a smallest support set can be arbitrarily larger than the size (depth) of an optimal DT for that classification instance (Theorem 18).*

For the bounded domain case (all features are binary), we obtain the following results.

- *The smallest size (depth) of a DT for a binary classification instance using only features of a smallest support set is at most by an exponential factor larger than the size (depth) of an optimal DT for that classification instance (Theorem 16).*
- *There exist binary classification instances where this exponential factor is unavoidable (Theorem 17).*

These separation results are relevant to practitioners who develop algorithms for DT minimization. It is tempting to first minimize the set of features to achieve a smaller instance size, so that the input to a SAT or CP encoding is easier to handle. However, our separation results establish that one has to consider that the result will be significantly worse than the optimum.

2 Preliminaries

We give some basic definitions of Parameterized Complexity and refer for a more in-depth treatment to other sources [Cygan *et al.*, 2015; Downey and Fellows, 2013]. PC considers problems in a two-dimensional setting, where a problem instance is a pair (I, k), where I is the main part and k is the parameter. A parameterized problem is *fixed-parameter tractable* if there exists a computable function f such that instances (I, k) can be solved in time $f(k)\|I\|^{O(1)}$.

2.1 Classification Problems

An *example* e is a function $e : \mathit{feat}(e) \to \mathbb{Z}$ defined on a finite set $\mathit{feat}(e)$ of *features*, where each feature f comes with a possibly infinite linearly ordered domain $\mathit{dom}(f) \subseteq \mathbb{Z}$, which we assume to be, w.l.o.g., a subset of the integers. For a set E of examples, we put $\mathit{feat}(E) = \bigcup_{e \in E} \mathit{feat}(e)$. We say that two examples e_1, e_2 *agree* on a feature f if $f \in \mathit{feat}(e_1)$, $f \in \mathit{feat}(e_2)$ and $e_1(f) = e_2(f)$. If $f \in \mathit{feat}(e_1)$, $f \in \mathit{feat}(e_2)$ but $e_1(f) \neq e_2(f)$, we say that the examples *disagree on* f.

A *classification instance (CI)* (also called a *partially defined Boolean function* [Ibaraki *et al.*, 2011]) $E = E^+ \uplus E^-$ is the disjoint union of two sets of examples, where for all $e_1, e_2 \in E$ we have $\mathit{feat}(e_1) = \mathit{feat}(e_2)$. The examples in E^+ are said to be *positive*; the examples in E^- are said to be *negative*. A set X of examples is *uniform* if $X \subseteq E^+$ or $X \subseteq E^-$; otherwise X is *non-uniform*. We say that a CI E is binary if all features in $\mathit{feat}(E)$ are *binary*, i.e., $e(f) \in \{0,1\}$ for every $f \in \mathit{feat}(e)$.

Given a CI E, a subset $F \subseteq \mathit{feat}(E)$ is a *support set* of E if any two examples $e_1 \in E^+$ and $e_2 \in E^-$ disagree in at least one feature of F. Finding a smallest support set, denoted by $\mathrm{MSS}(E)$, for a classification instance E is an NP-hard task [Ibaraki *et al.*, 2011, Theorem 12.2].

2.2 Decision Trees

A *decision tree* (DT) is a rooted binary tree T with vertex set $V(T)$ and edge set $A(T)$ such that each leaf node is either a *positive* or a *negative* leaf and the following holds for each non-leaf t of T:

- t is labelled with a feature denoted by $\mathit{feat}_T(t)$ or simply $\mathit{feat}(t)$ if T is clear from the context,
- t is labelled with an integer *threshold* denoted by $\lambda_T(t)$ or simply $\lambda(t)$ if T is clear from the context,
- t has 2 children, i.e., a left child and a right child.

We write $\mathit{feat}(T) = \{\, \mathit{feat}(t) \mid t \in V(T) \,\}$ for the set of all features used by T. The size of T is its number of nodes, i.e. $|V(T)|$.We denote by $\mathit{dep}(T)$ the depth of T, i.e., the maximum number of nodes on any root-to-leaf path on T.

Let E be a CI and let T be a DT with $\mathit{feat}(T) \subseteq \mathit{feat}(E)$. We say that a node t_A is a *left (right) ancestor* of t if t is contained in the subtree of T rooted at the left (right) child of t_A. For each node t of T, we define $E_T(t)$ as the set of all examples $e \in E$ such that for every left (right) ancestor t_A of t in T, it holds that $e(\mathit{feat}(t_A)) \leq \lambda(t_A)$ ($e(\mathit{feat}(t_A)) > \lambda(t_A)$). T *classifies* an example $e \in E$ if e is a positive (negative) example and $e \in E_T(l)$ for a positive (negative) leaf l of T. We say that T *classifies* E (or T *is a DT for* E) if T classifies all examples in E. See Figure 1 for an illustration of a CI and a DT that classifies E.

We will consider the following optimization problems.

E	f_1	f_2	f_3	f_4
$e_1 \in E^-$	0	5	1	-2
$e_2 \in E^-$	1	-1	3	0
$e_3 \in E^-$	1	0	-1	1
$e_4 \in E^-$	3	1	0	-1
$e_5 \in E^+$	4	-2	2	0
$e_6 \in E^+$	2	1	1	1

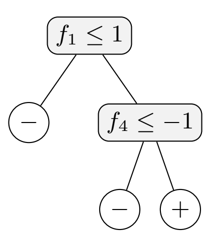

Figure 1: A CI $E = E^+ \uplus E^-$ with six examples and four features (left), a decision tree with 5 nodes that classifies E (right).

MINIMUM DECISION TREE SIZE (DTS)

Input: A CI E and an integer s.
Question: Find a DT of size at most s for E or report correctly that there is no DT for E of size at most s.

MINIMUM DECISION TREE DEPTH (DTD)

Input: A classification instance E and an integer d.
Question: Find a DT of minimum depth (or height) for E or report correctly that there is no DT for E of depth d.

For two examples e and e' in E, we denote by $\delta(e, e')$ the set of features where e and e' disagree and we denote by $\delta_{\max}(E) = \max_{e^+ \in E^+ \wedge e^- \in E^-} |\delta(e^+, e^-)|$ the *maximum difference* between any non-uniform pair of examples.

Let T be a DT for E and $t \in V(T)$ be an inner node of T. We denote by T_t the (sub-)DT of T rooted at t. We say that t is *redundant* if either: (1) t is the root of T and either T_{c_ℓ} or T_{c_r} is a DT for E, where c_ℓ and c_r are the left and right children of t in T, or (2) t is the left (right) child of its parent p and t has a child c such that the tree obtained from T after removing T_c and t and making the other child of t the left (right) child of p is a DT for E. Intuitively, t is redundant if it is not required to distinguish any examples and can therefore be removed from T. We say that T is *non-redundant* if it does not contain any redundant node.

For the complexity analysis we set the *input size* $\|E\|$ of a CI E to $|E| \cdot (|\mathit{feat}(E)| + 1) \cdot \log D_{\max}$, where $D_{\max}$ is the maximum size of $\mathit{dom}(f)$ over all features f of E. We now give some simple auxiliary lemmas that are required by our algorithms.

Observation 1 ([Ordyniak and Szeider, 2021, Obs. 1])**.** *Let T be a DT for a CI E, then $\mathit{feat}(T)$ is a support set of E.*

Lemma 2 ([Ordyniak and Szeider, 2021, Cor. 9])**.** *Let E be a CI and let k be an integer. Then there is an algorithm that in time $\mathcal{O}(\delta_{\max}(E)^k |E|)$ enumerates all (of the at most $\delta_{\max}(E)^k$) minimal support sets of size at most k for E.*

The following lemma follows naturally from [Ordyniak and Szeider, 2021, Lem. 5], we include a proof for completeness.

Lemma 3 ([Ordyniak and Szeider, 2021])**.** *Let A be a set of features of size a. Then the number of DTs without thresholds of size at most s that use only features in A is at most a^{2s+1} and those can be enumerated in $\mathcal{O}(a^{2s+1})$ time.*

Proof. We start by counting the number of trees T with n nodes that can potentially underlie a DT with n nodes. Note that there is one-to-one correspondence between trees T that underlie a DT with n nodes and unlabelled rooted ordered binary trees with n nodes (where ordered refers to an ordering of the at most 2 child nodes). Since it is known that the number of unlabelled rooted ordered binary trees with n nodes is equal to the n-th Catalan number C_n and that those trees can be enumerated in $\mathcal{O}(C_n)$ time [Stanley and Weisstein, 2015], we already obtain that we can enumerate all of the at most C_n possible trees T underlying a DT of size n in $\mathcal{O}(C_n)$ time. Therefore, there are at most sC_s possible trees of size at most s that can underlie a DT with at most s nodes and those can be enumerated in $\mathcal{O}(sC_s)$ time. It now remains to bound the number of possible feature assignments *feat*(f) for these trees as well as the number of possibilities for the leave nodes that can be either labelled positive or negative. Since we can assume that $a \geq 2$, we obtain that the number of possible feature assignments (and label lings of leaf-nodes) of a tree T with n nodes is at most a^n. Taking everything together, we obtain that there are at most $sC_sa^s \leq s4^sa^s \leq a^{2s+1}$ many DTs of size at most s using only features in A and those can be enumerated in $\mathcal{O}(a^{2s+1})$ time. ☐

3 FPT-algorithm

This section is devoted to a proof of our main result provided in the following theorem.

Theorem 4. DTS *and* DTD *are fixed-parameter tractable parameterized by the solution size and* $\delta_{\max}$.

To simplify the presentation and taking into account that the proof for DTD is almost identically to the proof for DTS, we will start by showing the result for DTS.

The overall structure of our algorithm is very similar to Algorithms 3 and 4 given in [Ordyniak and Szeider, 2021] and is illustrated in Algorithms 1 and 2. Namely, Algorithm 1 contains the main routine **minDT**, which given a CI E and an integer s outputs a minimum DT, i.e., a DT of minimum size, for E among all DTs of size at most s. To achieve this, the routine **minDT** first iterates over all minimal support sets of size at most s using Lemma 2. It then calls the routine **minDTS**, given in Algorithm 2, for every such minimal support set S to find a minimum DT T for E of size at most s such that $S \subseteq$ *feat*(T). Note that provided the correctness of **minDTS**, the correctness of **minDT** follows from Observation 1, because every DT for E must contain some minimal support set. Given E, s and a minimal support S, the routine **minDTS** computes a minimum DT T for E of size at most s such that $S \subseteq$ *feat*(T). The starting point (recursion start) of **minDTS** is the following lemma that allows to compute a minimum DT T for E of size at most s such that $S =$ *feat*(T).

Lemma 5 ([Ordyniak and Szeider, 2021, Theorem 4])**.** *Let E be a CI, $S \subseteq$ feat(E) be a support set for E, and let s be an integer. Then, there is an algorithm that runs in time $2^{\mathcal{O}(s^2)}\|E\|^{1+o(1)}\log\|E\|$ and computes a minimum DT among all DTs T with feat$(T) = S$ and $|T| \leq s$ if such a DT exists;otherwise* nil *is returned. Similarly, there is an algorithm that runs in time $\mathcal{O}((2^d)^{2^d}n^{1+o(1)}\log n)$ and computes a DT of minimum depth among all DTs T with feat$(T) = S$ and dep$(T) \leq d$ if such a DT exists;otherwise the algorithm return* nil.

After applying the above lemma to find a minimum DT T for E of size at most s such that $S =$ *feat*(T), the routine **minDTS** tries to find a minimum DT for E of size at most s that uses at least one feature outside of S. To achieve this the algorithm first computes a so-called (S, s)-branching set H, which informally is a "small" set of features such that every DT T for E of size at most s with $S \subsetneq$ *feat*(T) has to

Algorithm 1 Main method for finding a DT of minimum size.

Input: CI E and integer s
Output: DT for E of minimum size (among all DTs of size at most s) if such a DT exists, otherwise nil

function MINDT(E, s)
  $\mathcal{S} \leftarrow$ "set of all minimal support sets for E of size at most s using Lemma 2"
  $B \leftarrow$ nil
  for $S \in \mathcal{S}$ **do**
    $T \leftarrow$ MINDTS(E, s, S)
    if $(T \neq$ nil$)$ and $(B =$ nil or $|B| > |T|)$ **then**
      $B \leftarrow T$
  if $B \neq$ nil and $|B| \leq s$ **then**
    return B
  return nil

use at least one feature in H (see Subsection 3.1 for a formal definition of (S, s)-branching set). It then branches on every feature h in H and calls itself recursively for E, s, and $S \cup \{h\}$. The main ingredient of our algorithm compared to the algorithm given in [Ordyniak and Szeider, 2021], i.e., the FPT-algorithm for DTS if one additionally parameterizes by the maximum domain size of any feature, is the computation of the (S, s)-branching set, which we describe next.

Algorithm 2 Method for finding a DT of minimum size using at least the features in a given support set S.

Input: CI E, integer s, support set S for E with $|S| \leq s$
Output: DT of minimum size among all DTs T for E of size at most s such that $S \subseteq$ *feat*(T); if no such DT exists, nil

function MINDTS(E, s, S)
  $B \leftarrow$ "a minimum size DT for E of size at most s that uses exactly the features in S using Lemma 5"
  $H \leftarrow$ "a (S, s)-branching set $B(S, s)$ using Theorem 6"
  for $f \in H$ **do**
    $T \leftarrow$ MINDTS$(E, s, S \cup \{f\})$
    if $T \neq$ nil and $|T| < |B|$ **then**
      $B \leftarrow T$
  if $|B| \leq s$ **then**
    return B
  return nil

3.1 Computing Branching Sets

Here, we will show that we can compute a small branching set, which is the main novel and crucial ingredient for our FPT-algorithm. Before we formally define branching sets, we need the following notions.

Let E be a CI. We denote by ■ a new feature, which we call the *unknown feature*, i.e., ■ $\notin$ *feat*(E). A *DT pattern* is a DT T without thresholds that is allowed to use the unknown feature, i.e., *feat*$(T) \subseteq$ *feat*$(E) \cup \{$■$\}$. We say that an inner node t of T is *known* if *feat*$(t) \in$ *feat*(E) and *unknown* otherwise. A DT pattern T' is an *extension* of a DT pattern T if $T = T'$ and *feat*$_{T'}(t) =$ *feat*$_T(t)$ for every known node t of T. We say that T' is *complete* if *feat*$(T') \subseteq$ *feat*(E). A *threshold assignment* for a DT pattern T is a function $\lambda : \mathsf{KN}(T) \rightarrow \mathbb{Z}$ that provides a threshold assignment for every node of T in the set $\mathsf{KN}(T)$ of all known nodes of T.

In the following, let T be a DT pattern for a CI E. Note that we assume that if t is a node of T with $feat(t) = \blacksquare$, then any example that ends up in t is sent to both its left and its right child in T. In particular, we generalize $E_T(t)$ to DT patterns T with a threshold assignment λ by setting $E_T(t)$ to be the set of all examples $e \in E$ such that for every left (right) ancestor t_A of t in T, it holds that either $feat(t_A) = \blacksquare$ or $e(feat(t_A)) \leq \lambda(t_A)$ ($e(feat(t_A)) > \lambda(t_A)$).

We say that a node t of T is *valid* for a set $E' \subseteq E$ of examples if there is threshold assignment $\lambda : \mathsf{KN}(T) \to \mathbb{Z}$ such that either:

- t is a negative (positive) leaf of T and $E' \subseteq E^-$ ($E' \subseteq E^+$), or
- t is an unknown node of T and t has a child t' that is valid for E', or
- t is a known node of T with feature $f = feat(t)$ and the two children c_l and c_r of t in T are valid for $E'[f \leq \lambda(t)]$ and $E'[f > \lambda(t)]$, respectively.

We also say that T is *valid* for E' if the root r of T is valid for E'. Intuitively, T is valid for E' if it can be completed to a DT for E' that does not use of any of the unknown nodes.

Let E be a CI and let T be an invalid DT pattern for E. We say that a set $B \subseteq feat(E) \setminus feat(T)$ is a *branching set* for T if $B \cap (feat(T') \setminus feat(T)) \neq \emptyset$ for every proper extension T' of T that is valid for E. Let s be an integer and let S be a support set for E with $|S| \leq s$. We say that a set $B \subseteq feat(E) \setminus S$ is an (S, s)*-branching set* if $B \cap (feat(T) \setminus S) \neq \emptyset$ for every non-redundant DT T for E of size at most s with $S \subsetneq feat(T)$.

The remainder of this subsection is devoted to a proof of the following theorem, which constitutes the main novel technical contribution of this paper and we believe is interesting in its own right.

Theorem 6. *Let s be an integer, E be a CI and S be a support set for E with $|S| \leq s$. Then, an (S, s)-branching set of size at most $(s + 3)^{2s+1}\delta_{\max}(E)$ and can be computed in time $\mathcal{O}((s + 1)^{2s+1}2^{s^2/2}\|E\|^{1+o(1)} \log \|E\|)$.*

The main ideas behind the proof of Theorem 6 are as follows. Given s, E, and S as defined in Theorem 6 our aim is to find a small set B of features, i.e., an (S, s)-branching set, such that $B \cap (feat(T) \setminus S) \neq \emptyset$ for every non-redundant DT T for E of size at most s such that $S \subsetneq feat(T)$. Let T be any such non-redundant DT for E, then replacing every feature in $feat(T) \setminus S$ with the new feature $\blacksquare$ and ignoring the threshold function gives rise to an invalid DT pattern T' for E; T' is invalid because T is non-redundant. The main ingredient behind our algorithm is now a routine that given any invalid DT pattern T' computes a small branching set for T'. Because an (S, s)-branching set can be obtained from the union of all branching sets for every possible invalid DT patterns for E of size at most s that uses only features in $S \cup \{\blacksquare\}$, this now allows us to compute an (S, s)-branching set as follows. First we use the following corollary of Lemma 3 to enumerate all possible DT patterns T' for E of size at most s using only features in $S \cup \{\blacksquare\}$.

Corollary 7. *Let A be a set of features of size a with $\blacksquare \in A$. The number of DTs patterns of size at most s that use only features in A is at most a^{2s+1} and those can be enumerated in $\mathcal{O}(a^{2s+1})$ time.*

We then use Lemma 10 to decide whether T' is valid for E. Finally, if this not the case we use our routine to compute a branching set for T'. The (S, s)-branching set is then obtained as the union of all branching sets computed in this manner. Therefore, our main task now is to compute a branching set for a given invalid DT pattern for E.

Let E be a CI and let T be an invalid DT pattern for E. Our algorithm to compute a branching set for T proceeds in two main steps. First we compute a set EXP_t of expected examples for every node t of T, which intuitively contains all examples that: (1) will end up at t if no unknown node is replaced with a real feature and (2) is the smallest set of examples showing that T is invalid. Second, given EXP_t we compute an even smaller subset of examples, i.e., a so called pool set $P(r)$ for the root r of T, satisfying (1) and (2). We then show that any valid extension of T has to replace at least one unknown feature with a feature that distinguishes between two examples in the pool set. This then allows us to show that the set of all features $\bigcup_{e,e' \in P(r)} \delta(e, e')$ is a branching set for T. We start by showing how we compute the set of expected examples.

Computing the Set of Expected Examples

Let E be a CI and T be an invalid DT pattern for E. For every $t \in V(T)$, we define the set of *expected examples* EXP_t together with the *left and right thresholds*, denoted by $\lambda^L(t)$ and $\lambda^R(t)$, respectively, recursively as follows:

- if t is the root of T, then $\mathsf{EXP}_t = E$;
- if t is the left child of a known node p, then $\mathsf{EXP}_t = \mathsf{EXP}_p[f \leq \lambda^L(p) + 1]$, where $f = feat(p)$ and $\lambda^L(p)$ is the maximum value in $dom(f)$ such that T_t is valid for $\mathsf{EXP}_t[f \leq \lambda^L(p)]$;
- if t is the right child of a known node p, then $\mathsf{EXP}_t = \mathsf{EXP}_p[f > \lambda^R(p) - 1]$ where $f = feat(p)$ and $\lambda^R(p)$ is the minimum value in $dom(f)$ such that T_t is valid for $\mathsf{EXP}_t[f > \lambda^R(p)]$;
- if t is a child of an unknown node p, then $\mathsf{EXP}_t = \mathsf{EXP}_p$.

Before proving in Theorem 11 that we can efficiently compute EXP_t, $\lambda^L(t)$, and $\lambda^R(t)$ for every (fixed) node t of T, we need to show some simple but crucial properties.

Lemma 8. *Let T be an invalid DT pattern for E. For every node t of T it holds that T_t is not valid for EXP_t.*

Proof. Let T be an invalid DT pattern for E. We show the statement by induction on the *depth* of the node t, i.e., the distance of t to the root of T, in T. The statement clearly holds if t has depth 0, i.e., t is the root of T, by the definition of validity. Therefore, towards showing the induction step, suppose that the statement holds for the parent p of t in T, i.e., T_p is not valid for EXP_p. We need to show that T_t is not valid for EXP_t. We distinguish the following cases: (1) p is an unknown node of T and (2) p is a known node of T with feature $f = feat(p)$. In the former case, assume for a contradiction that T_t is valid for EXP_t. Therefore, by the definition of validity for p, we obtain that T_p is valid for EXP_t and therefore also for EXP_p (because $\mathsf{EXP}_t = \mathsf{EXP}_p$ by the definition of EXP_t). However, this contradicts our assumption that T_p is invalid for EXP_p.

In the latter case (i.e., case (2)), suppose that t is the left child of p (the case that t is the right child of p is analogous) and suppose for a contradiction that T_t is valid for $\mathsf{EXP}_t = \mathsf{EXP}_p[f \leq \lambda^L(p) + 1]$ and let $\lambda : \mathsf{KN}(T_t) \to \mathbb{Z}$ be the threshold assignment for T_t witnessing this. But then, because of the definition of $\lambda^L(p)$, it holds that $\lambda^L(p)$ is equal to the maximum domain value of f and therefore, it holds that T_t is also valid for EXP_p. However, this implies that T_p is valid for the threshold assignment obtained from λ after setting $\lambda(p)$ to the maximum domain value of f contradicting our assumption that p is invalid for EXP_p. ☐

Lemma 9. *Let T be an invalid DT pattern for E. For every known node t of T it holds that $\lambda^L(t) < \lambda^R(t)$.*

Proof. Let T be a DT pattern that is not valid for E. Suppose for a contradiction that there is a known node t with feature $f = \mathit{feat}(t)$ such that $\lambda^L(t) \geq \lambda^R(t)$. Let c_l and c_r be the left and right child of t in T. By the definition of $\lambda^L(t)$ T_{c_l} is valid for $\mathsf{EXP}_t[f \leq \lambda^L(t)]$ and therefore there is a threshold assignment $\lambda_L : \mathsf{KN}(T_{c_l}) \to \mathbb{Z}$ for T_{c_l} witnessing this. Similarly, by the definition of $\lambda^R(t)$ T_{c_r} is valid for $\mathsf{EXP}_t[f > \lambda^L(t)]$ and therefore there is a threshold assignment $\lambda_R : \mathsf{KN}(T_{c_r}) \to \mathbb{Z}$ for T_{c_r} witnessing this. But then then threshold assignment $\lambda : \mathsf{KN}(T_T) \to \mathbb{Z}$ obtained from $\lambda_L \cup \lambda_R$ after setting $\lambda(t)$ to $\lambda^L(t)$, shows that T_t is valid for EXP_t, contradicting the fact that t is invalid for EXP_t by Lemma 8. ☐

Algorithm 3

Input: CI E, DT pattern T for E
Output: `TRUE` if T is valid for E, `FALSE` otherwise

function ISVALID(E, T)
    $r \leftarrow$ "root of T"
    if r is a leaf **then**
        if r is negative (positive) and $E \subseteq E^-$ ($E \subseteq E^+$) **then**
            return `TRUE`
        return `FALSE`
    $c_\ell, c_r \leftarrow$ "left child and right child of r"
    if r is unknown **then**
        if ISVALID(E, T_{c_ℓ}) or ISVALID(E, T_{c_r}) **then**
            return `TRUE`
        return `FALSE`
    $f \leftarrow \mathit{feat}(r)$
    $(\lambda^L, \lambda^R) \leftarrow$ BINARYSEARCH(E, T, f, c_ℓ, c_r)
    if $\lambda^L \geq \lambda^R$ **then**
        return `TRUE`
    return `FALSE`

The following lemma, which is a precursor for the computation of the expected examples in Theorem 11, is a relatively straightforward extension of [Ordyniak and Szeider, 2021, Lemma 6]; the algorithm behind the lemma is also illustrated in Algorithms 3 and 4.

Algorithm 4 Algorithm to compute the pair $(\lambda^L(r), \lambda^R(r))$ for the root r of T

Input: CI E, DT pattern T, feature f of the root of T, left child c_ℓ of the root of T, right child c_r of the root of T
Output: the pair $(\lambda^L(r), \lambda^R(r))$

function BINARYSEARCH(E, T, f, c_ℓ, c_r)
    $D \leftarrow$ "array containing all elements in $\mathit{dom}_E(f)$ in ascending order"
    $L \leftarrow 0$; $R \leftarrow |D| - 1$;
    while $L \leq R$ **do**
        $m \leftarrow \lfloor (L+R)/2 \rfloor$
        if ISVALID($E[f \leq D[m]]$, T_{c_ℓ}) **then**
            $L \leftarrow m + 1$;
        else
            $R \leftarrow m - 1$;
    $\lambda^L \leftarrow D[m-1]$ ▷ where $D[-1] = D[0] - 1$
    $L \leftarrow 0$; $R \leftarrow |D| - 1$;
    while $L \leq R$ **do**
        $m \leftarrow \lfloor (L+R)/2 \rfloor$
        if ISVALID($E[f > D[m]]$, T_{c_r}) **then**
            $R \leftarrow m - 1$;
        else
            $L \leftarrow m + 1$;
    $\lambda^R \leftarrow D[m+1]$ ▷ where $D[|D|] = D[|D|-1] + 1$
    return (λ^L, λ^R)

Lemma 10. *Let E be a CI and T be a DT pattern of depth at most d. There is an algorithm with run-time $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)} \log \|E\|)$ deciding whether T is valid for E.*

Proof. Let E be a CI and let T be a DT pattern of depth at most d. In order to verify whether T is valid for E we have to attempt to find a threshold assignment $\lambda : \mathsf{KN}(T) \to \mathbb{Z}$ that is a witness to the validity of T. We prove that we can verify the validity of the root r of T by induction on the depth of T. Let us consider the base case, i.e., r is also a negative (positive) leaf (and the unique node) of T. By definition, it is enough to check whether E is a subset of E^- (E^+).

Therefore, towards showing the induction step, suppose that T is a DT pattern of depth at least one and that for the two children c_ℓ and c_r of r there is an algorithm that runs in time $\mathcal{O}(2^{(d-1)^2/2} n^{1+o(1)} \log n)$ ($n = \|E\|$) and decides whether c_ℓ and c_r are valid for E and, in the case the check is successful, it outputs threshold assignments λ_{c_ℓ} for T_{c_ℓ} and λ_{c_r} for T_{c_r}. We distinguish the following cases: (1) r is an unknown node of T and (2) r is a known node of T with feature $f = feat(r)$. In the former case, it is enough to run the algorithm that test the validity of c_ℓ and of c_r for E. If either c_ℓ or c_r turn out to be valid for E, say for example c_ℓ is valid for E with threshold assignment $\lambda_{c_\ell} : \mathsf{KN}(T_{c_\ell}) \to \mathbb{Z}$, then r is valid for E too: λ_{c_ℓ} is also a witness of the validity of r, since r is unknown. Otherwise, i.e., if both c_ℓ and c_r are not valid for E then also r is not valid for E.

In the latter case (i.e., case (2)), the task is to understand whether it is possible to find an integer $\lambda(r)$ such that c_ℓ and c_r are valid for $E[f \leq \lambda(r)]$ and $E[f > \lambda(r)]$, respectively. The idea is to run a binary search on $\mathit{dom}(f)$ that outputs two integers: λ^L is the maximum integer such that c_ℓ is valid for

$E[f \leq \lambda^L]$ and, in a similar manner, λ^R is the minimum integer such that c_r is valid for $E[f > \lambda^R]$. Note that λ^L and λ^R always exist at the cost of considering any element smaller or larger than any element in $dom(f)$, respectively. The algorithm now compares the values of λ^L and λ^R. If $\lambda^L \geq \lambda^R$ then it is possible to combine the thresholds assignments λ_{c_ℓ} for T_{c_ℓ} and λ_{c_r} for T_{c_r} to a threshold assignment λ for T: the threshold assignment $\lambda_{c_\ell} \cup \lambda_{c_r} \cup \{\lambda^L\}$ is a witness of the validity of r. Suppose otherwise $\lambda^L < \lambda^R$: this means that for any integer λ^*, either c_ℓ or c_r are not valid for $E[f \leq \lambda^*]$ or $E[f > \lambda^*]$, respectively. By definition, we can conclude that in this case r is not valid for E.

A key element of this algorithm for the known node case is a binary search sub-routine. This sub-routine attempts to find extremal values λ^L and λ^R for which the nodes c_ℓ and c_r are valid for $E[f \leq \lambda^L]$ and $E[f > \lambda^R]$, respectively. Every time this sub-routine calls the algorithm, it corresponds to check the validity of the same DT pattern but with a different set of examples. Understanding the extremal values for λ^L and λ^R is crucial: only the comparison between the extremal values of λ^L and λ^R allows to certify the correctness of our approach and algorithm.

The overall idea is to use algorithm **isValid** illustrated in Algorithm 3. That is, given a CI E and a DT pattern T, the algorithm **isValid** attempts to check whether T is valid for E. In Lines 3 to 6, the algorithm deals with the case where T has depth 0 and so its root r is also a leaf: it returns `TRUE` if $E \subseteq E^-$ if r is negative ($E \subseteq E^+$ if r is positive) and `FALSE` otherwise.

Starting from Line 7 to the end of the algorithm, the cases where is r is not a leaf node are analysed. In Lines 8 to 11, the algorithm deals with the case where r is an unknown node: here there are two recursive calls that attempt to check whether c_ℓ and c_r are valid for E. The algorithm returns `TRUE` if there is at least one `TRUE` output and `FALSE` otherwise.

Finally, in Lines 13 to 15, the algorithm deals with the case where r is a known node and let g be the feature of r in T. There is a call to **binarySearch** which is outlined in Algorithm 4. Given a CI E, a DT pattern T, the feature f and the left and right child of r, c_ℓ and c_r, this sub-routine performs a standard binary search procedure on the array D containing all the values in $dom_E(f)$ in ascending order to find the maximum threshold λ^L and minimum threshold λ^R such that T_{c_ℓ} and T_{c_r} are valid for $E[f \leq \lambda^L]$ and for $E[f > \lambda^R]$ respectively. To achieve this, the sub-routine makes at most $\log |E|$ calls to **isValid**; note that each of those calls is made for a tree of smaller depth. Lines 3 to 10: the sub-routine finds the maximum λ^L by calling algorithm **isValid** in Line 6 repeatedly. Lines 11 to 18: the algorithm finds the minimum λ^R by calling algorithm **isValid** in Line 14 repeatedly.

The running time of Algorithm 3 can now be obtained by multiplying the number of recursive calls to **isValid** with the time required for one recursive call. To obtain the number of recursive calls first note that if **isValid** is called with DT pattern of depth d, then it makes at most $(2 \log n) + 2$ recursive calls to **isValid** with a pattern of depth at most $d - 1$, where $n = \|E\|$. Therefore the number $T(n, d)$ of recursive calls for a pattern of depth d is given by the recursion relation $T(n, d) = (2(\log n) + 2)T(n, d - 1)$ starting with $T(n, 0) = 0$. This implies that $T(n, d) \in \mathcal{O}((\log n)^d)$. Finally, the run-time for one recursive call is easily seen to be at most $\mathcal{O}(n \log n)$. Hence, the total run-time of the algorithm is at most $\mathcal{O}((\log n)^d n \log n)$, which because (see also [Cygan *et al.*, 2015, Exercise 3.18]):

$$(\log n)^d \leq 2^{d^2/2} 2^{\log \log d^2/2} = 2^{d^2/2} n^{o(1)}$$

is at most $\mathcal{O}(2^{d^2/2} n^{1+o(1)} \log n)$. □

Algorithm 5 Algorithm to compute the triple $(\mathsf{EXP}_t, \lambda^L(t), \lambda^R(t))$ for every node $t \in V(T)$.

Input: CI E, DT pattern T
Output: the triple $(\mathsf{EXP}_t, \lambda^L(t), \lambda^R(t))$ for every node $t \in V(T)$.

```
function FINDLR(E, T)
    r ← "root of T"
    if r is a leaf then
        return (E, nil, nil)
    c_ℓ, c_r ← "left child and right child of r"
    if r is an unknown node then
        O_ℓ ← FINDLR(E, T_{c_ℓ})
        O_r ← FINDLR(E, T_{c_r})
        return (E, nil, nil) ∪ O_ℓ ∪ O_r
    f ← feat(r)
    (λ^L, λ^R) ← BINARYSEARCH(E, T, f, c_ℓ, c_r)
    O_ℓ ← FINDLR(E[f ≤ λ^L + 1], T_{c_ℓ})
    O_r ← FINDLR(E[f > λ^R − 1], T_{c_r})
    return (E, λ^L, λ^R) ∪ O_ℓ ∪ O_r
```

Now we are finally ready to prove that we can efficiently compute EXP_t, $\lambda^L(t)$ and $\lambda^R(t)$ for every node $t \in V(T)$.

Theorem 11. *Let E be a CI, let T be a DT pattern of depth at most d. Then there is an algorithm that runs in time $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)} \log \|E\|)$ and computes the set EXP_t and thresholds $\lambda^L(t)$ and $\lambda^R(t)$ for every node $t \in V(T)$.*

Proof. Let E be a CI and let T be a DT pattern of depth at most d. We prove we can compute the triple $(\mathsf{EXP}_t, \lambda^L(t), \lambda^R(t))$ by induction on the depth of the node t, i.e., the distance of t to the root of T, in T. Note that we are required to compute the left and right thresholds for a node only if it is a known node. For this reason, when considering a node t that is either unknown or a leaf, it is required only to compute the corresponding set of expected examples EXP_t and then return the triple $(\mathsf{EXP}_t, \mathtt{nil}, \mathtt{nil})$.

Let us consider the base case, i.e., t is the root of T. For EXP_t we can set it equal to E according to the definition. Suppose that t is also a known node with feature $f = feat(t)$: to conclude this case, we need to correctly compute the left and right threshold for t. The idea is to run a binary search on $dom(f)$, like we did for the algorithm of Lemma 10, that outputs two integers: $\lambda^L(t)$ is the maximum integer such that T_{c_ℓ} is valid for $E[f \leq \lambda^L(t)]$ and, in a similar manner, $\lambda^R(t)$ is the minimum integer such that T_{c_r} is valid for $E[f > \lambda^R(t)]$.

Therefore, towards showing the induction step, suppose that T is a DT pattern of depth at least one and that t is not the root of T. Let p be the parent of t in T. By the inductive hypothesis, we know that there is an algorithm that computes the triple $(\mathsf{EXP}_p, \lambda^L(p), \lambda^R(p))$. First thing we do is running this algorithm to obtain the triple $(\mathsf{EXP}_p, \lambda^L(p), \lambda^R(p))$.

Given $(\mathsf{EXP}_p, \lambda^L(p), \lambda^R(p))$ we can now compute EXP_t by distinguishing the following cases: If p is an unknown node, we set $\mathsf{EXP}_t = \mathsf{EXP}_p$. Moreover, if p is a known node with feature $f = \mathit{feat}(p)$, we set EXP_t to $\mathsf{EXP}_p[f \leq \lambda^L(p) + 1]$, if t is the left child of p, and we set EXP_t to $\mathsf{EXP}_p[f > \lambda^R(p) - 1]$ otherwise. Given EXP_t it only remains to show how to compute $\lambda^L(t)$ and $\lambda^R(t)$ if t is a known node with feature $f_t = \mathit{feat}(t)$ and children c_ℓ and c_r. Note that by definition $\lambda^L(t)$ is the maximum threshold such that T_{c_ℓ} is valid for $\mathsf{EXP}_t[f_t \leq \lambda^L(t)]$ and $\lambda^R(t)$ is the minimum threshold such that T_{c_r} is valid for $\mathsf{EXP}_t[f_t > \lambda^R(t)]$. Therefore, we can use the **binarySearch** function defined in Algorithm 4 called with parameters EXP_t, T_t, f_l, c_ℓ, and c_r to compute the pair $(\lambda^L(t), \lambda^R(t))$.

The overall idea stems from the recursive algorithm **findLR** illustrated in Algorithm 5. Given a CI E and a DT pattern T, the algorithm **findLR** returns the triple $(\mathsf{EXP}_r, \lambda^L(r), \lambda^R(r))$ for root node r of T and call itself for the children of r (if r is not a leaf) to compute the corresponding triples. In Lines 3 to 4, algorithm **findLR** deals with the case r is a leaf (and so it is the unique node) of T. Since r does not have children, the left and right threshold for r are directly set as `nil`.

Starting from Line 5 to the end of the algorithm, the cases where r is not a leaf node a analysed. In Lines 6 to 9, the algorithm deals with the case where r is an unknown node: here there is a recursive call to compute the corresponding triple for each of the two children of r. Since r is an unknown node, the left and right threshold for r are directly set as `nil` and the triple $(E, \texttt{nil}, \texttt{nil})$ is returned.

In Lines 10 to 14, the algorithm deals with the case where r is a known node: first there a is a call to the **binarySearch** sub-routine in Line 11 that outputs the left and right thresholds λ^L and λ^R for r. Then, there is a recursive call to compute the corresponding triple for each of the two children of r. Finally the triple $(E, \lambda^L, \lambda^R)$ is returned. A key element for the correctness of Algorithm 5 is that every time the algorithm call itself on the DT rooted at a child of the current node, the correct set of expected examples for that child is passed as part of the input of that recursive call. For this is the reason, the set E does not get updated during the current call.

The running time analysis for Algorithm 5 is exactly the same as the one for Algorithm 3 as the structure of the two algorithms is basically the same. $\square$

Computing the Pool and Branching Set

Let E be a CI and T a DT pattern for E that is invalid for E and suppose that we have already computed the triple $(\mathsf{EXP}_t, \lambda^L(t), \lambda^R(t))$ for every node $t \in V(T)$. We say that T' is a *proper* extension of T if T' is an extension of T and $\mathit{feat}_{T'}(t) \notin \mathit{feat}(T)$ for every unknown node $t \in V(T)$ with $\mathit{feat}_{T'}(t) \neq \blacksquare$, i.e., unknown nodes of T that are known in T' are assigned to features not in $\mathit{feat}(T)$.

A *pool set* for T is a set P of examples such that for every proper extension of T that is valid for E there is a feature $f \in \mathit{feat}(T') \setminus \mathit{feat}(T)$ such that f distinguishes two examples in P. Let $P(t)$ be the set of examples defined recursively for every node t of T as follows: If t is a negative (positive) leaf node of T, then $P(t)$ contains any example in $E^+ \cap \mathsf{EXP}_t$ ($E^- \cap \mathsf{EXP}_t$). Note that such an example does always exists because of Lemma 8 and our assumption that T is invalid for E. Otherwise, t has a left child c_ℓ and a right child c_r and we set $P(t) = P(c_\ell) \cup P(c_r)$. Note that $P(t) \subseteq \mathsf{EXP}_t$ for every $t \in V(T)$. We show next that $P(T) = P(r)$ for the root r of T is a pool set for T.

Lemma 12. *Let E be a CI and T be an invalid DT pattern for E. Then, $P(T)$ is a pool set for T.*

Proof. Assume for a contradiction that this is not the case. Then, there is a proper extension T' of T that is valid for E such that no feature in $\mathit{feat}(T') \setminus \mathit{feat}(T)$ distinguishes between any two examples in $P(T)$. Let $\lambda : \mathsf{KN}(T') \to \mathbb{Z}$ be a threshold assignment for T' showing the validity of T'. We start by showing the following claim:

Claim 13. *Let t be an inner node of T such that $P(t) \subseteq E_{T'}(t)$, then t has a child c in T such that $P(c) \subseteq E_{T'}(c)$.*

Proof. Let c_ℓ and c_r be the left and right child of t in T, respectively.

First consider the case that t is known in T and let $f = \mathit{feat}_T(t)$. Because T is not valid for E, we obtain from Lemma 9 that $\lambda^L(t) < \lambda^R(t)$ and therefore $f(e_\ell) \leq f(e_r)$ for every two examples with $e_\ell \in \mathsf{EXP}_{c_\ell}$ and $e_r \in \mathsf{EXP}_{c_r}$. Therefore, no matter the value of $\lambda(t)$ either all examples in $P(c_\ell) \subseteq \mathsf{EXP}_{c_\ell}$ are send to c_ℓ (and therefore $P(c_\ell) \subseteq E_{T'}(c)$) or all examples in $P(c_r) \subseteq \mathsf{EXP}_{c_r}$ are send to c_r (and therefore $P(c_r) \subseteq E_{T'}(c)$), which shows the claim.

Now consider the case that t is unknown in T and let $f = \mathit{feat}_{T'}(t)$. If $f = \blacksquare$, then $E_{T'}(c_\ell) = E_{T'}(c_r) = E_{T'}(t)$ and therefore the claim obviously holds. Otherwise, we know that f does not distinguish between any two examples in $P(t)$ and therefore either $P(t) \subseteq E_{T'}(c_\ell)$ or $P(t) \subseteq E_{T'}(c_r)$, which because $P(c_\ell), P(c_r) \subseteq P(t)$ implies the statement of the claim. $\square$

Because the conditions of Claim 13 apply to the root r of T, it follows that T must have a leaf l with $P(l) \subseteq E_{T'}(l)$. But this implies that T'_l is not valid for $E_{T'}(l)$ a contradiction to our assumption that T' is valid for E. $\square$

The next lemma shows that $P(T)$ is indeed small and can be computed efficiently.

Lemma 14. *Let E be a CI and T be an invalid DT pattern for E of height at most d. Then, $P(T) \leq 2^d$ and $P(T)$ can be computed in time $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)} \log \|E\|)$.*

Proof. $P(T) \leq 2^d$ follows because $|P(l)| = 1$ for every leaf of T and $|P(t)| = |P(c_\ell)| + |P(c_r)| = 2|P(c_\ell)|$ for every inner node t with children c_ℓ and c_r. To compute $P(T)$, we first use Theorem 11 to compute the triple $(\mathsf{EXP}_t, \lambda^L(t), \lambda^R(t))$ for every node $t \in V(T)$ in time $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)} \log \|E\|)$. We then compute $P(T)$ in a leaf-to-root manner in time $(|V(T)|)$. $\square$

The next lemma now show that the set $B(T) = \bigcup_{e,e' \in P(T)} \delta(e,e')$ is a branching set for T, i.e., we can easily compute a branching set from a pool set.

Lemma 15. *Let E be a CI and T be an invalid DT pattern for E of height at most d. Then, $B(T)$ is a branching set for T of size at most $2^{2d}\delta_{\max}(E)$ and can be computed in time $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)}\log\|E\|)$.*

Proof. $B(T)$ is a branching set because $P(T)$ is a pool set for T due to Lemma 12. Moreover, because of Lemma 14, we have that $|B| \leq |P(T)|^2\delta_{\max}(E) \leq 2^{2d}\delta_{\max}(E)$ and the time required to compute $P(T)$ is $\mathcal{O}(2^{d^2/2}\|E\|^{1+o(1)}\log\|E\|)$, which dominates the time to compute $B(T)$. □

We are now ready to show Theorem 6, i.e., we will show that $B(S,s) = \bigcup_{T \in \mathcal{T}} B(T)$ is a small (S,s)-branching set and can be efficiently computed, where $\mathcal{T}$ is the set of all invalid DT patterns for E of size at most s using only features in $S \cup \{\blacksquare\}$.

Proof of Theorem 6. We start by showing that $B(S,s)$ is an (S,s)-branching set. Let T be any non-redundant DT for E of size at most s such that $S \subsetneq \mathit{feat}(T)$ and let T' be the DT pattern for E obtained from T after setting $\mathit{feat}_{T'}(t) = \blacksquare$ for every $t \in V(T)$ with $\mathit{feat}_T(t) \notin S$ and ignoring all thresholds. Because T' has at least one unknown node and T is non-redundant, it follows that T' is invalid for E. Therefore, $T' \in \mathcal{T}$, which shows that $B(T') \subseteq B(S,s)$. Because $B(T')$ is a branching set for T' and T is a proper extension of T' that is valid for E, we obtain that $B(T') \cap (\mathit{feat}(T) \setminus S) \neq \emptyset$ and therefore $B(S,s)$ is an (S,s)-branching set, as required.

Towards showing how to compute $B(S,s)$, let $\mathcal{T}'$ be the set of all DT patterns for E of size at most s that use only features in $S \cup \{\blacksquare\}$. Because of Lemma 3, the set $\mathcal{T}'$ can be computed in time $\mathcal{O}((|S|+1)^{2s+1}) = \mathcal{O}((s+1)^{2s+1})$. Moreover, because of Lemma 10, we can decide whether T is invalid for E for every $T \in \mathcal{T}'$ in time at most $\mathcal{O}(2^{s^2/2}n^{1+o(1)}\log n)$, where $n = \|E\|$. Together this allows us to compute the set $\mathcal{T}$ from $\mathcal{T}'$ in time $\mathcal{O}((s+1)^{2s+1}2^{s^2/2}n^{1+o(1)}\log n)$. Finally, because of Lemma 15, we can compute $B(T)$ for every $T \in \mathcal{T}$ in time $\mathcal{O}(2^{s^2/2}n^{1+o(1)}\log n)$, which shows that $B(S,s)$ can be computed in time $\mathcal{O}((s+1)^{2s+1}2^{s^2/2}n^{1+o(1)}\log n)$.

Finally, $B(S,s)$ has size at most $|\mathcal{T}|$ times the maximum size of $|B(T)|$ over all $T \in \mathcal{T}$, which because of Lemma 15 is at most $2^{2d}\delta_{\max}(E) \leq 2^{2s}\delta_{\max}(E)$. Since $|\mathcal{T}| \leq |\mathcal{T}'| \leq (s+1)^{2s+1}$ (because of Lemma 3), we obtain that $|B(S,s)| \leq (s+1)^{2s+1}2^{2s}\delta_{\max}(E) \leq (s+3)^{2s+1}\delta_{\max}(E)$. □

We are now ready to show Theorem 4, i.e., that DTS is fixed-parameter tractable parameterized by size and $\delta_{\max}$.

Proof. Our algorithm for DTS is illustrated in Algorithm 1 and Algorithm 2.

Given a CI E and an integer s, the algorithm, given by the function **minDT** in Algorithm 1, returns a DT for E of minimum size among all DTs of size at most s if such a DT exists and otherwise the algorithm returns `nil`. The algorithm starts by computing the set $\mathcal{S}$ of all minimal support sets for E of size at most s with the help of Lemma 2. The main ingredient for the algorithm is the function **minDTS** illustrated in Algorithm 2 that the algorithm calls in Line 5 for every support set S in $\mathcal{S}$. Given E, s, and S the function **minDTS** returns a DT of minimum size among all DTs T for E of size at most s such that $S \subseteq \mathit{feat}(T)$ if such a DT exists and `nil` otherwise. It then updates the currently best DT B if necessary with the DT found by the function **minDTS**. Moreover, if the best DT found after iterating over all supports sets in $\mathcal{S}$ has size at most s, it is returned (in Line 9), otherwise the algorithm returns `nil`. Finally, the function **minDTS** illustrated in Algorithm 2 does the following. It first computes a DT T of minimum size that uses exactly the features in S using Lemma 5. It then tries to improve upon T with the help of an (S,s)-branching set H, which it computes in Line 3 with the help of Theorem 6. That is, the algorithm now iterates over every feature f in H and calls itself recursively for the support set $S \cup \{f\}$ in Line 5 in order to decide whether adding f gives rise to a smaller DT. If this call finds a smaller DT, then the current best DT is updated. Finally, after iterating over all features in H, the algorithm either returns the current best DT B if its size is at most s or `nil` otherwise.

We are now ready to show the correctness of Algorithm 1. So suppose that there is a DT for E of size at most s that uses all features in S and let T be any such DT of minimum size. Because the algorithm returns a DT of minimum size among all the DTs that it considers, it suffices to show that the algorithm considers T. Even stronger we will show that the algorithm considers all DTs T' for E of size at most s such that $\mathit{feat}(T') = \mathit{feat}(T)$.

Towards showing the correctness of Algorithm 1, consider the case that E has a DT of size at most s and let T be such a DT of minimum size. Because of Observation 1, $\mathit{feat}(T)$ is a support set for E and therefore $\mathit{feat}(T)$ contains a minimal support set S of size at most s. Because the algorithm (Line 4 of Algorithm 1) iterates over all minimal support sets of size at most s for E, it follows that Algorithm 2 is called with parameters E, s, and S.

If $\mathit{feat}(T) = S$, then the algorithm finds a DT for E of size at most $|T|$ in Line 2 of Algorithm 2 because of Lemma 5. If, on the other hand, $\mathit{feat}(T) \setminus S \neq \emptyset$, then $H \cap \mathit{feat}(T) \neq \emptyset$, where H is the (S,s)-branching set computed in Line 3 of Algorithm 2; this is because H is an (S,s)-branching set and T is a non-redundant (since minimal) DT for E of size at most s such that $S \subsetneq \mathit{feat}(T)$. Therefore, the function **minDTS** is called recursively for parameters E, s, and $S \cup \{f\}$, where f is an arbitrary feature in $\mathit{feat}(T) \cap H$. From now onward the argument repeats and eventually the function **minDTS** is called with parameters E, s, and $\mathit{feat}(T)$ at which point the algorithm finds a DT for E of size at most $|T|$ in Line 2 of Algorithm 2. Finally, it is easy to see that if Algorithm 1 outputs a DT T, then it is a valid solution. This is because T must have been computed in Line 2 of Algorithm 2, which implies that T is a DT for E. Moreover, T has size at most s, because of Line 8 in Algorithm 1.

To analyse the run-time of the algorithm, we first remark that the whole algorithm can be seen as a bounded-depth search tree algorithm, i.e., a branching algorithm with small recursion depth and few branches at every node. In particular, every recursive call adds at least one feature to the set of features bounding the recursion depth to at most s. Moreover, every feature that is added is either added in Line 2 of Algorithm 1, when enumerating all minimal support sets, in which case there are at most $\delta_{\max}(E)$ branches or the feature is added in Line 5 of Algorithm 2, in which case there are at most $|H| \leq (s+3)^{2s+1}\delta_{\max}(E)$ branches. It follows that the algorithm can be seen as a branching algorithm of depth at most s with at most $\max\{(s+3)^{2s+1}\delta_{\max}(E), \delta_{\max}(E)\} = (s+3)^{2s+1}\delta_{\max}(E)$ branches at every step.

Therefore, the total run-time of the algorithm is at most the number of nodes in the branching tree, i.e., at most $((s+3)^{2s+1}\delta_{\max}(E))^s$, times the maximum time required in one recursive call. Now the maximum time required for one recursive call is dominated by the time spend in Line 2 of Algorithm 2, i.e., the time required to compute a DT of minimum size using exactly the features in S with the help of Lemma 5, which is at most $\mathcal{O}(s^{2s+1}2^{s^2/2}n^{1+o(1)}\log n)$, where $n = \|E\|$. Therefore, we obtain $\mathcal{O}(((s+3)^{2s+1}\delta_{\max}(E))^s s^{2s+1}2^{s^2/2}n^{1+o(1)}\log n)$ as the total run-time of the algorithm, which shows that DTS is fixed-parameter tractable parameterized by $s + \delta_{\max}(E)$.

Algorithm 6 Main method for finding a DT of minimum depth.

Input: CI E and integer d
Output: DT for E of minimum depth (among all DTs of depth at most d) if such a DT exists, otherwise `nil`

function MINDTD(E, d)
    $\mathcal{S} \leftarrow$ "set of all minimal support sets for E of size at most 2^d using Lemma 2"
    $B \leftarrow$ `nil`
    for $S \in \mathcal{S}$ **do**
        $T \leftarrow$ MINDTDS(E, d, S)
        if ($T \neq$ `nil`) and ($B =$ `nil` or $|B| > |T|$) **then**
            $B \leftarrow T$
    if $B \neq$ `nil` and *dep*(B) $\leq d$ **then**
        return B
    return `nil`

Algorithm 7 Method for finding a DT of minimum depth using at least the features in a given support set S.

Input: CI E, integer d, support set S for E with $|S| \leq 2^d$
Output: DT of minimum depth among all DTs T for E of depth at most d such that $S \subseteq$ *feat*(T); if no such DT exists, `nil`

function MINDTS(E, s, S)
    $B \leftarrow$ "a minimum depth DT for E of depth at most d that uses exactly the features in S using Lemma 5"
    $H \leftarrow$ "a $(S, 2^d)$-branching set $B(S, 2^d)$ using Theorem 6"
    for $f \in H$ **do**
        $T \leftarrow$ MINDTDS(E, d, $S \cup \{f\}$)
        if $T \neq$ `nil` and $|T| < |B|$ **then**
            $B \leftarrow T$
    if *dep*(B) $\leq d$ **then**
        return B
    return `nil`

The algorithm for DTD is essentially very similar and the details are provided in Algorithm 6 that uses Algorithm 7 as a sub-routine. One of the main differences is that instead of searching for a set of features of size at most s, we now search for a set of features of size at most 2^d. This also has an influence on the run-time. The ideas behind the algorithm as well as the proof of correctness are, however, very similar. $\square$

4 Approximation Using Support Sets

Given Observation 1 it is tempting to think that it sufficies to consider only DTs that use the features from some minimal support set. Indeed, if this were the case, then our FPT-algorithm from the previous section could be significantly simplified, i.e., it would no longer be necessary to find branching sets as it would suffice to enumerate all minimal support sets with the help of Lemma 2. Unfortunately, Ordyniak and Szeider [2021] showed that this is not the case and the difference between an optimal DT and an optimal DT that is only allowed to employ features from some minimal support set can be arbitrarily high at least in absolute terms even for binary CIs. Nevertheless, it was left open whether and how well the simple approach using only minimal support sets can be exploited to obtain good approximate solutions for DTS and DTD and this is what we will explore in this section. In particular, let $\mathsf{opt}^s(E)$ and $\mathsf{opt}^d(E)$ be the minimum size respectively depth of a DT for a CI E and let $\mathsf{opt}^s_{\mathrm{SS}}(E)$ and $\mathsf{opt}^d_{\mathrm{SS}}(E)$ the minimum size respectively depth of a DT for E that is only allowed to use the features from some minimal support set. Because $\mathsf{opt}^s_{\mathrm{SS}}(E)$ and $\mathsf{opt}^d_{\mathrm{SS}}(E)$ can be computed using a much simpler algorithm that requires only Lemma 2 and Lemma 5, we want to explore whether they can be used to approximate $\mathsf{opt}^s(E)$ and $\mathsf{opt}^d(E)$.

As a starting point consider the case of binary CIs. In particular, let E be a binary CI and let S be a minimum support set for E. Then, because of Observation 1 any DT for E has size at least $|S|$ and depth at least $\log|S|$. Moreover, E has a DT of size at most $2^{|S|+1}$ and depth at most $|S|+1$, i.e., the complete DT using only the features in S. Therefore, we obtain the following theorem showing that $\mathsf{opt}^s_{\mathrm{SS}}$ and $\mathsf{opt}^d_{\mathrm{SS}}$ approximate opt^s and opt^d, respectively, for binary CIs.

Theorem 16. *Let E be a binary CI. Then, $\mathsf{opt}^s_{\mathrm{SS}}(E) \leq 2^{\mathsf{opt}^s(E)}$ and $\mathsf{opt}^d_{\mathrm{SS}}(E) \leq 2^{\mathsf{opt}^d(E)}$.*

As our main novel result in this section (for binary CIs), we show next that the ratios obtained in Theorem 16 are indeed best possible and therefore no better approximation for DTS and DTD can be obtained by considering only DTs that merely use the features of some minimal support set.

Theorem 17. *For every integer $k \geq 1$, there is a binary CI L_k such that $\mathsf{opt}^s(L_k) \leq 2k+5$ and $\mathsf{opt}^s_{\mathrm{SS}}(L_k) \geq 2^{k+1}-1$. Similarly, there is a binary CI L^d_k such that $\mathsf{opt}^d(L^d_k) \leq \log(k)+2$ and $\mathsf{opt}^d_{\mathrm{SS}}(L^d_k) \geq k+1$.*

Finally, we consider the case of non-binary CIs and show the following theorem, which essentially rules out any approximation algorithm based solely on minimal support sets.

Theorem 18. *For every integer $n \geq 1$, there are a CIs L_n and L^d_n such that $\mathsf{opt}^s(L_n), \mathsf{opt}^d(L^d_n) \leq 5$ and $\mathsf{opt}^s_{\mathrm{SS}}(L_n), \mathsf{opt}^d_{\mathrm{SS}}(L^d_n) \geq n$.*

For convenience the proofs for Theorem 17 and Theorem 18 are provided in the Sections 5 respectively 6 below.

5 Proof of Theorem 17

This section is devoted to a proof of Theorem 17. The proof is split into two main parts, i.e., we show the theorem for the case of size (the existence of L_k) in Subsection 5.1 and we show the theorem for the case of depth (the existence of L_k^d) in Subsection 5.2. Since we will only deal with binary CIs, we can assume that $\lambda(t) = 0$ for every inner node t of a DT and we will therefore omit the threshold function for simplicity.

We start by introducing the complete binary CI E_k on k features since it is required in both subsections. For every natural number $k \geq 1$, let E_k be the complete binary CI on k features, i.e., E_k has k features $f_1, \dots, f_k$ and one example for every of the 2^k possible assignments of those features. We denote by S_k the set of features $\{f_1, \dots, f_k\}$ of E_k. Moreover, an example $e \in E_k$ is positive $|\{ f \in S_k \mid e(f) = 1 \}|$ is even and negative otherwise.

5.1 Size

Here we show Theorem 17 for the case of size, i.e., we show the existence of the CIs L_k for every $k \geq 1$. Namely, let L_k be the CI obtained from E_k after adding a new feature f^* defined as follows. Let D_k be the set of all the examples $e \in E_k$ such that $e(f_i) = 1$ for every $i \in [k-2]$ and let $\overline{D_k}$ be the set $E_k \setminus D_k$ of all remaining examples. Then, we set $e(f^*) = 1$ if either e is a positive example or $e \in D_k$ and $e(f^*) = 0$ otherwise. Refer also to Figure 2 (left) for a visual representation of L_3 and the decomposition into D_3 and $\overline{D_3}$.

f_1	f_2	f_3	f^*		
0	0	0	1	+	
0	0	1	0	−	$\overline{D_3}$
0	1	0	0	−	
0	1	1	1	+	
1	0	0	1	−	
1	0	1	1	+	
1	1	0	1	+	D_3
1	1	1	1	−	

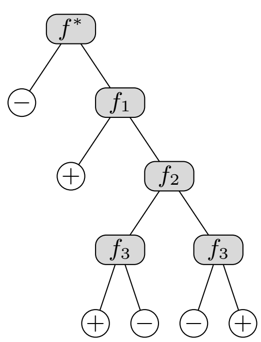

Figure 2: The CI L_3 partitioned into D_3 and $\overline{D_3}$ (left), the DT T_3 (right).

We start by showing that S_k is the only minimal support set for L_k.

Lemma 19. *Let $k \geq 1$ be an integer. Then, S_k is the only minimal support set for L_k.*

Proof. First, we note that by construction S_k is clearly a support set for E_k and therefore also for L_k. Therefore, it only remains to show that for every $i \in [k]$, the set $S_k^i = \mathit{feat}(L_k) \setminus \{f_i\}$ is not a support set for L_k.

For the case that $i \in [k-2]$, let e_i^+ any positive example in $\overline{D_k}$ with $e_i^+(f_i) = 0$ and let e_i^- be the unique negative example in D_k agreeing with e_i^+ on all features in $S_k^i \setminus \{f^*\}$. Then, $e_i^-(f^*) = e_i^+(f^*) = 1$ and therefore the two examples cannot be distinguished by any feature in S_k^i.

Otherwise, i.e., if $i \in [k-1, k]$, let e^+ be any positive example in D_k and let e^- be the unique negative example in D_k that differs from e^+ only at feature f_i. Then, $e_i^-(f^*) = e_i^+(f^*) = 1$ and therefore the two examples cannot be distinguished by any feature in S_k^i. □

The next result shows that every (non-redundant) DT for L_k that uses only the features in the unique minimal support set S_k has necessarily the structure of a complete binary tree of large size and depth.

Lemma 20. *For every integer $k \geq 1$, a non-redundant DT T with features in S_k is a DT for L_k if and only if T is a complete DT of depth $k+1$. In particular, such a DT has size $2^{k+1} - 1$.*

Proof. In this proof we assume that a leaf is either positive or negative depending on the parity of the number of right arcs present in the unique path from the root to that leaf. We start with the forward direction: let T be a non-redundant DT that is not a complete DT of depth $k+1$. Let P be a path of T from the root to a leaf ℓ of length at most k: at most $k-1$ features appear in P and so there exists a feature $f_i \in S_k$ that does not appear in P. Since by Lemma 19 $S_k^i = \mathit{feat}(L_k) \setminus \{f_i\}$ is not a support set for L_k, there exits a negative example e^- and a positive example e^+ that can not be distinguished by S_k^i, this means that $\{e^-, e^+\} \subseteq E_T(\ell)$ and so T is not a DT for L_k.

In order to prove the reverse direction, we assume that T is a non-redundant and complete DT of depth $k+1$ with features in S_k. Let P be a path of T from the root to a leaf ℓ; note that P is of length $k+1$. Since T is non-redundant, every feature of S_k appears exactly once in P. Since, by Lemma 19, S_k is a support set, there is only one example e_ℓ that ends ℓ, that is $\{e_\ell\} = E_T(\ell)$.

From this proof, it follows that every non-redundant DT T with features in S_k for L_k has $2^{k+1} - 1$ nodes. □

We now show that L_k has a DT of size at most $2k + 5$, i.e., the DT T_k that is constructed as follows. The root r of T_k has feature f^*. The left child c_ℓ of r is a negative leaf and the right child v_1 has feature f_1. For every $i \in [k-2]$, the left child of v_i is a positive leaf and the right child v_{i+1} has feature f_{i+1}. Finally v_k and v_k' are respectively the left and right child of v_{k-1}, both having feature f_k. The children of v_k and v_k' are leaves that are either positive or negative depending on the parity of the number of right arcs present in the unique path from the root to that leaf. See Figure 2 (right) for a visual representation of T_3 and note that T_k has $2k+5$ nodes. We show next that T_k is a DT for L_k.

Lemma 21. *For every integer $k \geq 1$, T_k is a DT for L_k.*

Proof. By construction, r and its feature f^* send every negative example to its left child c_ℓ, which is a negative leaf, except for the two negative examples in D_k, that is, if $\{e_1^-, e_2^-\} = E_k^- \cap D_k$, then $E_{T_k}(c_\ell) = E_k^- \setminus \{e_1^-, e_2^-\}$ and $E_{T_k}(v_1) = E_k^+ \cup \{e_1^-, e_2^-\}$.

Let e be an example in D_k; by construction, for every $i \in [k-2]$ if $e \in E_{T_k}(v_i)$ then $e \in E_{T_k}(v_{i+1})$ and by induction we obtain that $e \in E_{T_k}(v_{k-1})$. Let e be an example in $\overline{D_k}$ and $j \in [k-2]$ be the minimum integer such that $e(f_j) = 0$. This means that $e \notin E_{T_k}(v_{j+1})$ and e is classified by the left child of the node v_j. We have just proved that $D_k = E_{T_k}(v_{k-1})$ and that T_k classifies $\overline{D_k}$. Now it is straightforward to show that the subtree of T_k rooted at v_{k-1} classifies D_k. □

We are now ready to proof the first part of Theorem 17.

Proof of Theorem 17 (for size). By Lemma 19, S_k is the smallest (and unique minimal) support set for L_k and by Lemma 20, we have that every non-redundant DT for L_k that uses all and only the features in S_k has size at least $2^{k+1} - 1$. Moreover, since by Lemma 21 T_k is a DT for L_k, we have that the smallest DT for L_k has size at most $|T_k| = 2k + 5$. Therefore, $\mathsf{opt}^s(L_k) \leq 2k + 5$ and $\mathsf{opt}^s_{\mathsf{SS}}(L_k) \geq 2^{k+1} - 1$, as required. □

5.2 Depth

For every integer $k \geq 1$, let us describe a DT T^k as follows. The tree T^k has v_1 as root. For every $i \in [k]$, the node v_i has v_{2i} and v_{2i+1} as left child and right child, respectively. Moreover, the node v_i has feature f_i if $i \in [k]$, and is a leaf otherwise. A leaf ℓ of T^k is positive if the number of right arcs of the unique path from v_1 to ℓ is even and negative otherwise. Note that T^k has depth $\log(k) + 1$.

Let F_k be the set of all the examples in E_k that are correctly classified by T^k and denote by $\overline{F_k} = E_k \setminus F_k$. See Figure 3 for a visual representation of E_3 and its decomposition in F_3 and $\overline{F_3}$.

f_1	f_2	f_3	f'	f''		
0	0	0	0	0	+	
0	1	0	0	0	−	F_3
1	0	0	0	0	−	
1	0	1	0	0	+	
0	0	1	1	0	−	
0	1	1	1	1	+	
1	1	0	1	1	+	$\overline{F_3}$
1	1	1	1	0	−	

Figure 3: The CI E_3 partitioned into F_3 and $\overline{F_3}$

Let f' be a new feature defined as follows: $e(f') = 0$ if $e \in F_k$ and $e(f') = 1$ otherwise. We also define another new feature f'' as follows: $e(f'') = 0$ if either $e \in F_k$ or e is a negative example and $e(f'') = 1$ otherwise. Then, the CI L_k^d, whose existence is claimed in Theorem 17, is obtained from E_k after adding the two novel features f' and f'' and for simplicity, we denote by S'_k the set $\{f_1, \ldots, f_k, f', f''\}$.

We now introduce a DT of small depth for L_k^d, i.e., the DT T_*^k. For every integer $k \geq 1$, let T_*^k be the DT described as follows. The root r of T_*^k has feature f' and its left branch is the DT T^k. The right child of r is a node u with feature f''. The left/right child of u is a negative/positive leaf. Note that T_*^k has depth $\log(k) + 2$. See Figure 4 for a visual representation of the DTs T^3 (left) and T_*^3 (right).

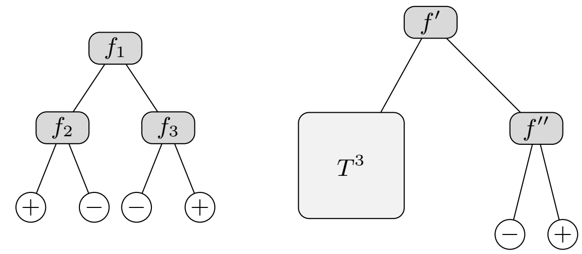

Figure 4: The DTs T^3 (left) and T_*^3 (right).

As for the case of size, we start by showing that S_k is the only minimal support set for L_k^d.

Lemma 22. *For every integer $k \geq 1$, S_k is the only minimal support set for L_k^d.*

Proof. By the proof of Lemma 19, the set S_k is a support set for E_k and therefore also for L_k^d. In the rest of the proof, we show that, for every $i \in [k]$, the set $S_k^i = S'_k \setminus \{f_i\}$ is not a support set for L_k^d, which completes the proof of the lemma.

Let e_i be the example of L_k^d described as follows: e_i is obtained from the minimal (partial) assignment that corresponds to the unique path from v_1 to v_i in T^k by setting all other values of the assignment to 0. First we prove that $e_i \in F_k$. By construction, the number of features $f \in S_k$ such that $e_i(f) = 1$ is equal to the number of right arcs of the unique path from v_1 to a leaf ℓ described by the assignment e_i: thus e_i and ℓ have the same positivity and so e_i is correctly classified by T^k. Considering the case $k = 3$ represented in Figure 3, it is easy to see that $e_1 = e_2 = (0, 0, 0, 0, 0)$ and $e_3 = (1, 0, 0, 0, 0)$.

Let e'_i be the example in L_k^d such that $e'_i(f_j) = e_i(f_j)$ for every $j \in [k] \setminus \{i\}$ and $e'_i(f_i) = 1 - e_i(f_i)$. Now we prove that (1) e'_i has different positivity than e_i and (2) $e'_i \in F_k$. To prove (1), it is enough to observe that e_i and e'_i differ on exactly one feature, f_i, and so the hamming distance between them is one: by definition, e_i and e'_i have different positivity.

In order to prove (2), it is enough to observe that the number of features $f \in S_k$ such that $e'_i(f) = 1$ is equal to the number of right arcs of the unique path from v_1 to a leaf ℓ described by the assignment e'_i: thus e'_i and ℓ have the same positivity and so e'_i is correctly classified by T^k. Considering the case $k = 3$ represented in Figure 3, it is easy to see that $e'_1 = (1, 0, 0, 0, 0)$, $e'_2 = (0, 1, 0, 0, 0)$ and $e'_3 = (1, 0, 1, 0, 0)$.

Thanks to the construction of e'_i, (1) and (2), we have shown that, for every $i \in [k]$, the pair e_i and e'_i is made of a positive and a negative example which can only be distinguished by feature f_i among those in S'_k: f_i must belong to every (minimal) support set for L_k^d. □

We now show that T_*^k is indeed a DT for L_k^d.

Lemma 23. *For every integer $k \geq 1$, T_*^k is a DT for L_k^d.*

Proof. By construction, r and its feature f' send every example of F_k to its left child and every other example, that is $\overline{F_k}$, to the right child. By definition, the set F_k is classified by

T^k and, by construction of f'', the subtree of T_*^k rooted at u classifies $\overline{F_k}$. Therefore, T_*^k classifies $F_k \cup \overline{F_k} = L_k^d$. □

We are now ready to proof the second part of Theorem 17.

Proof of Theorem 17 (for depth). By Lemma 19, S_k is the smallest (and unique minimal) support set for L_k^d and by Lemma 20, we have that every non-redundant DT for L_k^d that uses all and only the features in S_k has depth $k+1$. Moreover, since by Lemma 23 T_*^k is a DT for L_k^d, we have that the minimum depth of a DT for L_k^d is at most $depth(T_*^k) = log(k)+2$. Therefore, $\mathsf{opt}^d(L_k^d) \leq \log(k) + 2$ and $\mathsf{opt}^d_{\mathsf{SS}}(L_k^d) \geq k+1$, as required. □

6 Proof of Theorem 18

Here, we show Theorem 18. We start by introducing the classification instance L_k for every $k \geq 1$, whose existence is stated in the theorem. Let L_k be the CI with exactly k examples $\{e_1, \ldots, e_k\}$ on the 3 features f, f', and f'' defined as follows. For every $i \in [k]$, we set $e_i(f) = i$. Moreover, $e_i(f') = 1$ for every even $i \in [k-2]$ and $e_i(f') = 0$ otherwise. Finally, $e_i(f'') = 0$ for every odd $i \in [k-2]$ and $e_i(f') = 1$ otherwise. An example e_i is negative if i is odd and positive otherwise. See Figure 5 (left) for a visual representation of L_6.

f	f'	f''	
1	0	0	−
2	1	1	+
3	0	0	−
4	1	1	+
5	0	1	−
6	0	1	+

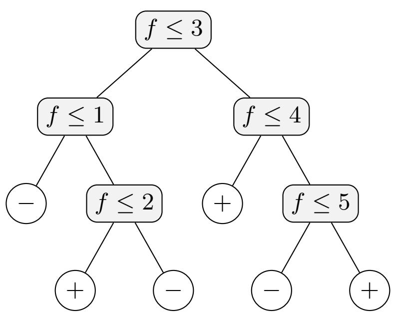

Figure 5: The CI L_6 (left) and the DT B_6 (right).

We start by showing that $\{f\}$ is the unique minimal support set for L_k.

Lemma 24. *For every integer $k \geq 1$, the set $\{f\}$ is the only minimal support set for L_k.*

Proof. First we note that $\{f\}$ is a support set for L_k: for every pair of positive and negative examples e_i and e_j for some even $i \in [k]$ and odd $j \in [j]$, feature f is able to distinguish e_i and e_j by choosing the threshold equal to $min\{i, j\}$.

It is also easy to see that $\{f', f''\}$ is not a support set for L_k: e_{k-1} and e_k have different parity and can not be distinguished by either f' or f''. □

For every integer $k \geq 1$, let us describe a DT B_k inductively as follows. Every internal node of B_k has feature f: we just have to describe the threshold chosen for such node. A leaf of B_k is positive if it is the left child of a node with even threshold or the right child of a node with odd threshold, and it is negative otherwise. B_1 is the DT with only one node. B_2 is the DT having only one internal node with threshold 1.

Now suppose we have all the DT B_i with $i < k$. The root of B_k has threshold $\lfloor \frac{k}{2} \rfloor$, the left branch is the DT $B_{\lfloor \frac{k}{2} \rfloor}$ and right branch is the DT $B_{\lceil \frac{k}{2} \rceil}$ but with all the thresholds increased by $\lfloor \frac{k}{2} \rfloor$. See Figure 5 (right) for a visual representation of B_6.

The next lemma shows how B_k is able to classify L_k.

Lemma 25. *For every integer $k \geq 1$, B_k has size $2k-1$, depth $\lceil \log(k) \rceil + 1$ and is a DT for L_k of minimum size and minimum depth among those that only use the feature f.*

Proof. We prove the statement by induction on k. For the base case, B_1 has just one (negative) node which trivially classifies L_1. Let us assume the statement is true for every integer $i < k$.

By construction, $|B_k| = 1 + |B_{\lfloor \frac{k}{2} \rfloor}| + |B_{\lceil \frac{k}{2} \rceil}| = 1 + (k-1) + (k-1) = 2k-1$ and $dep(B_k) = 1 + dep(B_{\lceil \frac{k}{2} \rceil}) = \lceil \log(k) \rceil + 1$.

Every example e_i, with $i \in [k]$, ends in either the left or right child of the root of B_k, depending on the comparison with $\lfloor \frac{k}{2} \rfloor$. By construction, the left/right child of the root of B_k is the root of the DT $B_{\lfloor \frac{k}{2} \rfloor}$/$B_{\lceil \frac{k}{2} \rceil}$ which classifies e_i by the inductive hypothesis.

Claim 26. *In every DT for L_k that uses only f as feature there is an internal node with threshold i, for every $i \in [k-1]$.*

Proof. Let B be a DT for L_k. Suppose, by contradiction, there exists an integer $i^* \in [k-1]$ that does not appear in an internal node of B as threshold. This means that e_{i^*} and e_{i^*+1} are not distinguished in B, which is a contradiction since they have different positivity. □

Suppose, by contradiction, that there exists a DT B_k^* for L_k, that uses only f as feature, of size smaller than $|B_k|$.

By Claim 26, B_k^* has $k-1$ internal nodes (as B_k does). As consequence, we have that B_k^* has less then k leaves: there is a leaf ℓ and integers $i, j \in [k-1]$, $i < j$ such that ℓ receives e_i and e_j. By how f is defined, if B_k^* can not distinguish e_i and e_j then it can not distinguish any pair of examples in $\{e_i, \ldots, e_j\}$; in particular B_k^* can not distinguish between e_i and e_{i+1}, which is a contradiction since they have different positivity. □

We are now ready to define the optimum DTs for L_k, which are allowed to use all features of L_k. For every integer $k \geq 1$, let C_k be the DT described as follows. The root r of C_k has feature f' with threshold 0 and its right child is a positive leaf c_r. The left child c_ℓ of r has feature f with threshold $k-1$: both children of c_ℓ are leaves; the left one is negative and the right one is positive.

Equivalently, for every integer $k \geq 1$, let C^k be the DT described as follows. The root r of C^k has feature f'' with threshold 0 and its left child is a negative leaf c_ℓ. The right child c_r of r has feature f with threshold $k-1$: both children of c_ℓ are leaves; the left one is positive and the right one is negative.

Note that, for every $k \geq 1$, C_k and C^k have 5 nodes. See Figure 6 for a visual representation of C_6 (left) and C^7 (right).

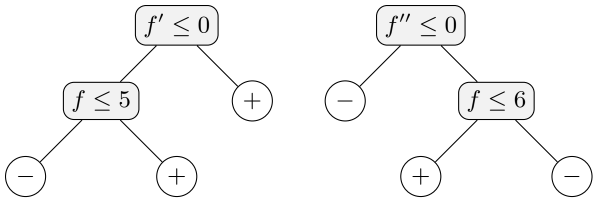

Figure 6: The DT C_6 (left) and C^7 (right).

Lemma 27. *For every even integer $k \geq 1$, C_k is a DT for L_k. Equivalently, for every odd integer $k \geq 1$, C^k is a DT for L_k.*

Proof. We prove the statement for even integers; the proof for odd integers is equivalent. Let k be an even integer. By construction, r, its feature f' and the threshold 0 sends all the positive examples to the right child, which is a positive leaf, except for e_k, that is, $E_{C_k}(c_\ell) = L_k^- \cup \{e_k\}$ and $E_{C_k}(c_r) = L_k^+ \setminus \{e_k\}$. The node c_ℓ, its feature f and its threshold $k-1$ can now distinguish L_k^+ and $\{e_k\}$, which allows to complete the classification of L_k. □

We are now ready to prove Theorem 18.

Proof of Theorem 18. By Lemma 24, $\{f\}$ is the smallest (and unique minimal) support set for L_k and by Lemma 25 we have that B_k is a DT for L_k of minimum size $2k-1$ and minimum depth $\lceil \log(k) \rceil + 1$ among those that only use the feature f. Moreover, by Lemma 27 either C_k or C^k is a DT for L_k that uses only $5 = O(1)$ nodes. Therefore, L_k satisfies $\mathsf{opt}^s(L_k) \leq 5$ and $\mathsf{opt}^s_{\mathsf{SS}}(L_k) \geq k$ and setting $L_k^d = L_{2^k}$ satisfies $\mathsf{opt}^s(L_k^d) \leq 5$ and $\mathsf{opt}^s_{\mathsf{SS}}(L_k^d) \geq k$, which completes the proof of the theorem. □

7 Conclusion

We have established novel results that contribute to the foundations of learning interpretable machine learning models. Our main result is algorithmic. We have devised a parameterized algorithm that allows us to efficiently learn an optimal DT (with the smallest number of nodes or lowest depth). The worst-case complexity of our algorithm depends on the input size and the combined parameter solution size, and the maximum difference. This answers an open question by Ordyniak and Szeider [2021], who had to include the maximum domain size for their FPT result and completes their complexity classification for DT learning. As pointed out in the introduction, our result stands out because for similar problems (like the CSP), the inclusion of domain size is inevitable.

Our second result deals with the question of what one loses when working with a smallest set of features (a minimum support set) when learning a DT of a small size or depth. It turns out that this question strongly depends on whether the domain size is bounded or not. We show that the gap between the optimal solution and one that depends on the smallest set of features can be arbitrarily large for the unbounded domain case. For the bounded domain case, the gap can be bounded by an exponential function, and that this bound is tight. This result is of interest to practitioners as it is a natural approach for heuristics to perform feature reduction before learning the DT.

Acknowledgements

Stefan Szeider acknowledges support by the Austrian Science Fund (FWF, project P32441) and the Vienna Science and Technology Fund (WWTF, project ICT19-065). Giacomo Paesani and Sebastian Ordyniak acknowledge support from the Engineering and Physical Sciences Research Council (EPSRC, project EP/V00252X/1).

References

[Aglin *et al.*, 2020a] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Learning optimal decision trees using caching branch-and-bound search. *Proc. AAAI 2020*, pages 3146–3153, 2020.

[Aglin *et al.*, 2020b] Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. PyDL8.5: a library for learning optimal decision trees. *Proc. IJCAI 2020*, pages 5222–5224, 2020.

[Avellaneda, 2020] Florent Avellaneda. Efficient inference of optimal decision trees. *Proc. AAAI 2020*, pages 3195–3202, 2020.

[Bäckström *et al.*, 2012] Christer Bäckström, Yue Chen, Peter Jonsson, Sebastian Ordyniak, and Stefan Szeider. The complexity of planning revisited - a parameterized analysis. *Proc. AAAI 2012*, (1):1735–1741, 2012.

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

[Bessière *et al.*, 2008] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, Claude-Guy Quimper, and Toby Walsh. The parameterized complexity of global constraints. *Proc. AAAI 2008*, pages 235–240, 2008.

[Bessiere *et al.*, 2009] Christian Bessiere, Emmanuel Hebrard, and Barry O'Sullivan. Minimising decision tree size as combinatorial optimisation. *Proc. CP 2009*, 5732:173–187, 2009.

[Bredereck *et al.*, 2017] Robert Bredereck, Jiehua Chen, Rolf Niedermeier, and Toby Walsh. Parliamentary voting procedures: Agenda control, manipulation, and uncertainty. *Journal of Artificial Intelligence Research*, 59:133–173, 2017.

[Cygan *et al.*, 2015] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[Darwiche and Hirth, 2023] Adnan Darwiche and Auguste Hirth. On the (complete) reasons behind decisions. *Journal of Logic, Language and Information*, 32(1):63–88, 2023.

[Demirovic *et al.*, 2022] Emir Demirovic, Anna Lukina, Emmanuel Hebrard, Jeffrey Chan, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Peter J.

Stuckey. MurTree: Optimal decision trees via dynamic programming and search. *Journal of Machine Learning Research*, 23(26):1–47, 2022.

[Doshi-Velez and Kim, 2017] Finale Doshi-Velez and Been Kim. A roadmap for a rigorous science of interpretability. *CoRR*, abs/1702.08608, 2017.

[Downey and Fellows, 2013] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer Verlag, 2013.

[Dvorák *et al.*, 2012] Wolfgang Dvorák, Sebastian Ordyniak, and Stefan Szeider. Augmenting tractable fragments of abstract argumentation. *Artificial Intelligence*, 186:157–173, 2012.

[Ganian *et al.*, 2018] Robert Ganian, Iyad Kanj, Sebastian Ordyniak, and Stefan Szeider. Parameterized algorithms for the matrix completion problem. *Proc. ICML 2018*, pages 1642–1651, 2018.

[Gaspers *et al.*, 2017] Serge Gaspers, Neeldhara Misra, Sebastian Ordyniak, Stefan Szeider, and Stanislav Zivný. Backdoors into heterogeneous classes of SAT and CSP. *Journal of Computer and System Sciences*, 85:38–56, 2017.

[Goodman and Flaxman, 2017] Bryce Goodman and Seth R. Flaxman. European union regulations on algorithmic decision-making and a “right to explanation”. *AI Magazine*, 38(3):50–57, 2017.

[Gottlob *et al.*, 2002] Georg Gottlob, Francesco Scarcello, and Martha Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1-2):55–86, 2002.

[Hu *et al.*, 2020] Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with MaxSAT and its integration in AdaBoost. *Proc. IJCAI 2020*, pages 1170–1176, 2020.

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

[Ibaraki *et al.*, 2011] Toshihide Ibaraki, Yves Crama, and Peter L. Hammer. *Partially defined Boolean functions (in Boolean Functions: Theory, Algorithms, and Applications)*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011.

[Janota and Morgado, 2020] Mikolás Janota and António Morgado. Sat-based encodings for optimal decision trees with explicit paths. *Proc. SAT 2020*, 12178:501–518, 2020.

[Larose and Larose, 2014] Daniel T. Larose and Chantal D. Larose. *Discovering Knowledge in Data: An Introduction to Data Mining*. John Wiley & Sons, 2nd edition, 2014.

[Lipton, 2018] Zachary C. Lipton. The mythos of model interpretability. *Communications of the ACM*, 61(10):36–43, 2018.

[Monroe, 2018] Don Monroe. AI, explain yourself. *AI Communications*, 61(11):11–13, 2018.

[Murthy, 1998] Sreerama K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[Narodytska *et al.*, 2018] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and Joao Marques-Silva. Learning optimal decision trees with SAT. *Proc. IJCAI 2018*, pages 1362–1368, 2018.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, Oxford, 2006.

[Ordyniak and Szeider, 2021] Sebastian Ordyniak and Stefan Szeider. Parameterized complexity of small decision tree learning. *Proc. AAAI 2021*, pages 6454–6462, 2021.

[Quinlan, 1986] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Samer and Szeider, 2010] Marko Samer and Stefan Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, 76(2):103–114, 2010.

[Schidler and Szeider, 2021] André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. *Proc. AAAI 2021*, pages 3904–3912, 2021.

[Shati *et al.*, 2021] Pouya Shati, Eldan Cohen, and Sheila A. McIlraith. SAT-based approach for learning optimal decision trees with non-binary features. *Proc. CP 2021*, 210(50):1–16, 2021.

[Stanley and Weisstein, 2015] Richard Stanley and Eric W. Weisstein. Catalan number, from mathworld–a wolfram web resource, 2015.

[Verhaeghe *et al.*, 2020] Hélène Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper, and Pierre Schaus. Learning optimal decision trees using constraint programming. *Constraints*, 25(3-4):226–250, 2020.

[Verwer and Zhang, 2017] Sicco Verwer and Yingqian Zhang. Learning decision trees with flexible constraints and objectives using integer optimization. *Proc. CPAIOR 2017*, 10335:94–103, 2017.

[Verwer and Zhang, 2019] Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. *Proc. AAAI 2019*, pages 1625–1632, 2019.

[Zhu *et al.*, 2020] Haoran Zhu, Pavankumar Murali, Dzung T. Phan, Lam M. Nguyen, and Jayant Kalagnanam. A scalable MIP-based method for learning optimal multivariate decision trees. *Proc. NeurIPS 2020*, 2020.