OPTIMAL HYPERSURFACE DECISION TREES

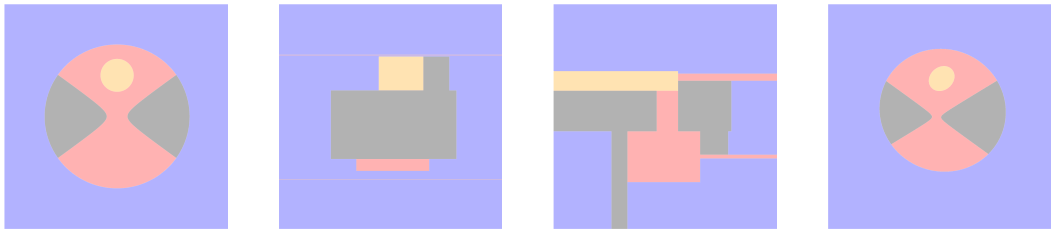
Anonymous authors
Paper under double-blind review

ABSTRACT

Optimal decision trees have attracted increasing attention in recent years, yet existing algorithms face two closely related challenges. First, the trees they construct have limited expressivity because their splitting rules are typically restricted to axis-parallel splits or binary features. Second, these algorithms generally scale poorly to large datasets. These challenges compound one another: more expressive splitting rules increase the combinatorial complexity of the search space, making the resulting optimal decision tree (ODT) problem even more difficult to solve.

Building on the proper decision tree framework of He & Little (2025b), we propose, to the best of our knowledge, the first algorithm for constructing optimal decision trees with hypersurface splits. The algorithm has worst-case work complexity $O\left(K!KN^{KG+1}\right)$ when $N \geq K$ and fixed-dimensional feasibility tests are treated as constant-cost operations, where $G = \binom{D+M}{M} - 1$ depends on the hypersurface polynomial degree M and the data dimension D. Its structure is naturally amenable to vectorization and parallelization, enabling efficient execution on GPUs, and the generic algorithm design can be used to accelerate other ODT variants, such as axis-parallel decision (ADT) trees. We further develop a safe pruning strategy that substantially reduces the number of candidate configurations and an incremental generation procedure that reduces the associated feasibility-checking cost from $O(K^2)$ to $O(K)$.

Experiments show that, for small trees, our efficient vectorized solver can explore over one million candidate trees in a matter of seconds. On both synthetic and real-world datasets, more expressive tree models consistently achieve higher predictive accuracy than approximate and optimal axis-parallel decision-tree models, often with substantially smaller tree sizes. In the most pronounced case, our model improves test accuracy by nearly 30% points relative to the optimal ADT algorithm.



(a) Ground truth
nmc: 0 d: 3, K: 3

(b) CART
nmc: 56 d: 5, K: 15

(c) ConTree
nmc: 34 d: 4, K: 15

(d) HSDT (ours)
nmc: 18 d: 3, K: 3

Figure 1: Synthetic dataset (left) generated by degree-2 polynomials; axis-parallel decision trees learned by CART and the state-of-the-art optimal algorithm—ConTree (middle two); and hypersurface decision trees (HSDT) learned using our algorithm (right), with corresponding numbers of misclassifications (nmc), tree depths (d), and tree sizes (K).

1 INTRODUCTION

Substantial recent advances have made optimal decision tree (ODT) more practical. Despite these advances, the term "optimal decision trees" is often used for several distinct tree definitions. In

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

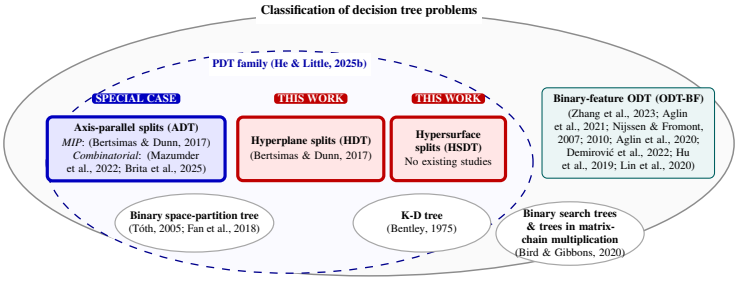


Figure 2: Taxonomy of optimal decision tree problems. Much of the prior work that claims to solve ODT problems solves only the ODT-BF problem. When solving ODT problems, one can optimize either *depth-constrained* trees or *size-constrained* trees (also referred to as *sparse decision trees*).

mainstream machine learning (ML), the classical ODT problem concerns axis-parallel decision trees (ADTs), which are commonly learned greedily using CART (Breiman et al., 1984). Finding an optimal ADT is NP-hard (Laurent & Rivest, 1976). However, only a small number of ODT studies directly address this problem (Bertsimas & Dunn, 2017; Mazumder et al., 2022; Brita et al., 2025); many methods described as ODT solvers instead address the simplified binary-feature ODT problem (ODT-BF) (Zhang et al., 2023; Demirović et al., 2022; Lin et al., 2020). For fixed tree size K, (He, 2025) shows that the ODT-BF problem is *polynomial-time solvable* in the sample size N and has been empirically observed to exhibit linear with respect to N (Hu et al., 2019). Figure 2 summarizes these classes and their relationships.

Existing ODT methods follow two main approaches. The first formulates the problem as mixed-integer programming (MIP). Since the pioneering work of Bertsimas & Dunn (2017), this approach has motivated substantial follow-up work (Boutilier et al., 2023; Günlük et al., 2021; Zhu et al., 2020; Verwer & Zhang, 2019). MIP formulations can flexibly accommodate diverse constraints and can, in some cases, solve large instances efficiently. However, their computational cost is often difficult to predict, and they may perform poorly even when a polynomial-time combinatorial solution exists.

The second approach uses *combinatorial methods*, including branch-and-bound (BnB) and dynamic programming (DP) algorithms (Brita et al., 2025; Aglin et al., 2021; Nijssen & Fromont, 2010; Aglin et al., 2020; Demirović et al., 2022; Hu et al., 2019; Lin et al., 2020; Mazumder et al., 2022). Much of this work focuses on the simpler ODT-BF setting. Moreover, its problem formulations are often informal and specified independently of a concrete algorithm. This separation creates a gap between theoretical characterizations and practical implementations: as reported by He & Little (2025b), material discrepancies exist between algorithmic abstractions and their actual implementations.

Moreover, existing studies still suffer from two major challenges:

- **Expressivity**: Classical decision trees have limited expressivity because they typically use axis-parallel splitting rules. As illustrated in Fig. 1, this restriction can produce a coarse approximation of the underlying ground truth. In the example, CART and the optimal axis-parallel tree learned using Brita et al. (2025)'s algorithm require 15 splitting rules and have depths of 4 and 5, respectively[1], yet neither closely captures the target partition.
- **Scalability**: More expressive tree models are especially difficult to scale. Even for hyperplane splits, the number of candidate splits grows as $O\left(N^D\right)$, far exceeding the $O\left(N \times D\right)$ candidates considered by classical ADTs. Existing DP methods (He, 2025; Mazumder et al., 2022; Brita et al., 2025) rely on *data-dependent recursion*[2] and depth-first search. These designs can have poor cache locality and are difficult to vectorize or parallelize effectively on modern hardware such as GPUs. Their pruning strategies are also often ad

[1] Additional comparisons in Fig. 6 (Appendix) show that a plausible fit is obtained only when CART uses 45 splitting rules and the optimal tree uses 48. Trees of this size are rarely practical in interpretable machine learning settings.

[2] *Data-dependent recursion* describes a recursive process whose number of calls, recursion depth, or control flow depends on the values or structure of the input data, rather than being fixed or statically predictable at compile time. In He (2025); Mazumder et al. (2022); Brita et al. (2025), the recursive structure of their algorithm depends on the dataset $\mathcal{D}$ and on the root selected in each branch.

hoc and specific to particular ODT variants. For example, the equivalent-points bound (Lin et al., 2020; Hu et al., 2019), which exploits duplicate data points, is useful for binary features but rarely applies to continuous data.

Consequently, researchers often rely on heuristics for large-scale problems or restrict attention to easier variants, such as ODT-BF. The combinatorial complexity of more flexible tree models, together with the lack of general pruning strategies and hardware-efficient algorithms, has limited their broader use.

In this paper, we address both challenges through the following contributions:

- **Exact formulation and algorithm.** We develop a *unified algorithmic framework* for axis-parallel, hyperplane, and hypersurface decision trees.
- **Hardware-friendly algorithmic design.** For the OHSDT problem, the number of candidate splits is $O\left(N^G\right)$, where $G = \binom{D+M}{M} - 1$ and M is the hypersurface polynomial degree. This search space is intractable even for greedy methods. We therefore introduce complementary acceleration strategies: a generic *vectorized algorithm* with improved hardware compatibility and parallelizability, while preserving the same theoretical complexity.
- **Safe pruning method and algorithmic fusion.** We introduce a novel *safe pruning method* that applies to both *HDTs and HSDTs* (but not ADTs), substantially reducing cost below the theoretical upper bound. We further exploit the prefix-closed infeasibility property to enable an *algorithmic fusion* that reduces the feasibility testing cost from $O(K^2)$ to $O(K)$.
- **Extensive empirical evaluation.** We evaluate on comprehensive synthetic and real-world datasets. On synthetic data, we assess robustness to variation and noise across **50** independently generated datasets per setting. The results show that, under controlled complexity, decision trees with more expressive splitting rules are more robust and achieve higher predictive accuracy.

2 Background—Proper decision tree framework

MIP definition of ODT problems The size-constrained ODT problem can be formulated using mixed-integer programming (MIP) as follows:

$$T^* = \operatorname*{argmin}_{T \in \mathcal{S}_{\text{size}}(K, xs)} E(T). \tag{1}$$

Here, E denotes the objective function; $\mathcal{S}_{\text{size}}(K, xs)$ denotes the search space of all decision trees with K internal nodes over the dataset xs; and T^* is an optimal decision tree.

Similar definitions appear in previous studies (Mazumder et al., 2022; Brita et al., 2025). Although the formulation is intuitive, constructs such as $\mathcal{S}_{\text{size}}(K, xs)$ and E are typically specified only through informal *textual descriptions* rather than given formal semantics. By contrast, Bertsimas & Dunn (2017) provide a rigorous MIP formulation of (1), but their formulation spans more than three pages.

We instead introduce a alternative *programmatic definition* of ODT problems that is both rigorous and concise.

Recursive definition of ODT problems In constructive algorithmics (Bird & De Moor, 1996), a problem may be specified directly as a program rather than first being defined as an algorithm-independent, informal mathematical object.

The programmatic formalization of ODT problems was first introduced by He (2025). They proved that any decision tree satisfying the axioms of a *proper decision tree (PDT)*—which ensure well-defined spatial partitions and transitive ancestry relations between splitting rules—admits two key results. First, the *PDT Characterization Theorem* (Theorem 6) characterizes PDTs by K*-permutations*. Second, the *PDT Decomposition Theorem* (Theorem 7) uses this characterization to decompose the ODT problem as follows:

$$odt\,(rs) = min_E \circ concatMapL\,(sodt\,(xs)) \circ kcombs_K\,(rs)\,. \tag{2}$$

Here, the algorithm odt takes a list of rules $rs : [\mathcal{R}]$ as input and returns an optimal decision tree t^*. On the right-hand side, the formulation derived by He (2025) comprises three functions, where the $\circ$ denotes function composition, defined by $f \circ g(a) = f(g(a))$. First, $kcombs_K(rs)$ generates all possible K-combinations of the rules in rs. Next, $concatMapL(sodt(xs))$ applies $sodt(xs)$ to each generated combination and concatenates the resulting lists. Finally, min_E selects an optimal solution according to the objective function E.

A key consequence of (2) is that the complex odt problem can be factorized into smaller **subproblems** generated by $kcombs_K(rs)$. Each corresponding $sodt$ instance can be solved **independently**, providing a natural basis for large-scale parallelization.

Size-/depth-constrained trees Most ODT solvers impose a **depth constraint**, whereas (1) and (2) optimize over **size-constrained trees**. Previous work has studied size-constrained trees primarily as *sparse decision trees* for ODT-BF (Hu et al., 2019; Lin et al., 2020). These two formulations induce different algorithmic structures. Specifically, He & Little (2025b) showed that the search space of an ODT method is determined by its **branching method** and **split generator**. Algorithms for trees in the PDT family share the same branching method but use different split generators depending on whether tree size or depth is constrained.

The choice between depth and size constraints entails distinct computational trade-offs. Depth constraints are more common in previous work and are often combined with *depth-first search*. This strategy can quickly explore large trees and identify accurate solutions *before* certifying optimality, but it typically exhibits poor cache locality and offers limited opportunities for parallelization. By contrast, through $kcombs$ in (2), the size-constrained search space decomposes naturally into subproblems that can be explored independently. Earlier sparse-tree methods have not exploited this decomposition. Because the hypersurface decision trees studied in this paper employ highly expressive splits, even small trees can often achieve sufficient predictive accuracy. We therefore focus on size-constrained trees and prioritize exploring many small trees in parallel.

On the other hand, because the two paradigms define different search spaces, direct comparison with existing depth-constrained methods is challenging. Nevertheless, the size-constrained algorithm of He & Little (2025b) provides the closest size-constrained counterpart to these methods because it uses the same branching method. We therefore use their algorithm as a baseline for evaluating the novel algorithms developed in this paper.

3 Optimal hypersurface decision tree algorithm

3.1 Rule generators for decision trees

Existing studies of ODT problems often use an ad hoc algorithm to solve ODT-BF (Demirović et al., 2022) and then develop a completely different algorithm to solve ADT problems (Brita et al., 2025). This problem-specific design increases the burden of understanding ODT problems, as readers must learn distinct algorithms for distinct ODT formulations.

Astute readers may note that our formulation of the ODT problem in (2) differs from the MIP formulation in (1): the former takes a list of rules rs as input, whereas the latter defines a search space over a dataset xs. This distinction is deliberate: it separates the *rule-generation program*, which produces the rules used to define decision trees, from the *program that constructs the trees*. Consequently, different ODT variants can be addressed by modifying only the rule-generation component while leaving the tree-construction process unchanged. To make this separation explicit, we define a function $genRules$ that generates rules from a given dataset. The complete procedure for solving an ODT problem is then defined as follows:

$$odt'\,(xs) = odt \circ genRules\,(xs)\,.$$

Here, $genRules : [\mathbb{R}^D] \to [\mathcal{R}]$ generates a set of rules from a dataset xs. The program odt' then solves (1) exactly; that is, it provides a programmatic definition of the ODT problem.

By modifying $genRules$, we can address different ODT variants while leaving the main program odt unchanged. This separation provides a modular formulation of ODT problems. For example, for

classical ADTs, $genRules$ generates all axis-aligned splits, whereas for hyperplane decision trees, it generates all possible hyperplanes.

At first sight, the latter task appears infeasible because the space of hyperplanes seems to have infinite combinatorial complexity: each hyperplane is defined by a continuous normal vector $\boldsymbol{w}_k$. Consequently, it may seem impossible for $genRules$ to terminate in a finite amount of time. However, the finiteness of the dataset imposes strong constraints. In particular, a finite dataset admits only finitely many distinct partitions induced by hyperplanes, thereby defining an **equivalence relation** among hyperplanes.

Indeed, following He & Little (2026)'s results, we can exhaustively enumerate the equivalence classes of hyperplanes and hypersurfaces using the following theorem.

Theorem 1. *Under the general-position assumption, the 0–1 loss (accuracy) linear classification problem on a dataset xs in $\mathbb{R}^D$ can be solved exhaustively by enumerating all D-point combinations of xs. Moreover, classification by a polynomial hypersurface[3] is isomorphic to classification by a hyperplane in a higher-dimensional feature space of dimension $G = \begin{pmatrix} M+D \\ M \end{pmatrix} - 1$, where M is the polynomial degree and D is the data dimension.*

This geometric result unifies the rule generators for different geometric splitting rules through the combination generator $genRules\,(G, xs) = kcombs\,(G, xs)$, parameterized by G.

Accordingly, odt' can be expanded as

$$odt'\,(xs) = min_E \circ concatMapL\,(sodt\,(xs)) \circ kcombs\,(K) \circ kcombs\,(G, xs)\,.$$

Interestingly, He & Little (2025a) show that the sequential composition of two combination generators, as in odt', can be fused into a single recursion, $nestedCombs\,(K, G, xs)$. This recursion is equivalent to $kcombs\,(K) \circ kcombs\,(G, xs)$ (see He & Little (2025a) for a proof and the imperative definition in Section 3.1). Therefore,

$$odt'\,(xs) = min_E \circ concatMapL\,(sodt\,(xs)) \circ nestedCombs\,(K, G, xs)\,. \tag{3}$$

Each nested combination generated by $nestedCombs\,(K, G, xs)$ is a K-combination of rules, with each rule constructed from a G-combination of data points. In the remainder of the paper, we use the terms *configurations* and *nested combinations* interchangeably.

3.2 Polynomial hypersurface decision trees

Theorem 1 establishes a natural isomorphism between hyperplane and polynomial-hypersurface classification. We may therefore state our results for hyperplanes, while retaining their applicability to hypersurfaces.

The following lemma establishes that decision trees defined by hyperplanes satisfy the PDT axioms.

Lemma 2. ***Hyperplane decision trees are proper.*** Let xs be a data set and let $hs_K = [h_1, h_2, \ldots, h_K]$ be a list of K hyperplanes with normal vectors $ws = [w_1, w_2, \ldots, w_K]$ in $\mathbb{R}^{D+1}$, where each hyperplane passes through exactly D data points. For each hyperplane, define the positive and negative regions by $h_i^+ = \left\{x \mid x \in \mathbb{R}^D, \boldsymbol{w}_i^T \bar{x} \geq 0\right\}$ and $h_i^- = \left\{x \mid x \in \mathbb{R}^D, \boldsymbol{w}_i^T \bar{x} < 0\right\}$. Then, decision trees constructed from these hyperplanes as splitting rules are proper decision trees.

By applying the PDT framework to the lifted dataset $\phi_M(xs) \subset \mathbb{R}^G$, where each polynomial splitting rule becomes an affine hyperplane. Pulling the resulting tree back through ϕ_M yields polynomial-hypersurface splits in the original space and preserves predictions and the 0–1 empirical objective, though not the topology of decision regions. Thus, He & Little (2025b)'s framework applies directly to HDTs and, through lifting, to HSDT problems.

However, the corresponding $sodt$ algorithm relies on data-dependent recursion, which limits parallelization and hardware compatibility. In the next section, we introduce a *compile-time-determined* recursion that enables full vectorization and efficient execution on modern hardware. This addresses a key gap in the original exposition of He (2025) and makes the odt program more amenable to parallelization. We denote the algorithm of He (2025) by $sodt_{\mathrm{rec}}$ and our new algorithm by $sodt_{\mathrm{vec}}$.

[3] A polynomial hypersurface is defined by a *polynomial equation* of the form $P\,(\mathbf{x}) = \sum_i w_i \mathbf{x}^{\alpha_i}$, where $\mathbf{x}$ is a D-dimensional data vector and $\boldsymbol{\alpha}$ is a multi-index exponent vector.

3.3 A NOVEL HARDWARE-FRIENDLY AND PARALLELIZABLE ALGORITHM

A size-constrained decision tree with K splitting rules has K internal nodes. Rather than recursively selecting a branch node and constructing its subtrees, as in $sodt_{rec}$, we incrementally add one internal node at a time until all K nodes have been introduced. This yields a deterministic process requiring at most K steps, so its recursion depth can be determined at compile time. Based on this observation, we define:

$$\begin{aligned} &genDTs_{\text{vec}} : \mathcal{D} \times [\mathcal{R}] \to [DTree\,(\mathcal{R}, \mathcal{D})] \\ &genDTs_{\text{vec}}\,(xs, [\,]) = [DL\,(xs)] \\ &genDTs_{\text{vec}}\,(xs, rs) = concat \circ [updates_{\boldsymbol{A}}\,(r, rs \backslash r, xs) \mid (r, rs \backslash r) \longleftarrow candidates\,(rs)] \end{aligned} \tag{4}$$

This recursion exhaustively generates all decision trees in the search space by appending a rule to the leaves of each partial tree through $updates_{\boldsymbol{A}}$. The complete recursive definition and imperative implementations appear in Appendix C.1.1 and Algorithm 1.

The optimal solution is obtained by selecting the best generated candidate:

$$sodt_{\text{vec}}\,(xs, rs) = min_E \circ genDTs_{\text{vec}}\,(xs, rs)$$

Both $sodt_{\text{vec}}$ and $sodt_{rec}$ have factorial dependence on K. The vectorized construction incurs an additional polynomial factor in sequential work, while exposing a regular batched computation that is more suitable for parallel hardware; see Theorem 9 for complexity analysis.

Our definition is fully formal and concise. For example, Figure 3 presents a four-rule example generated by a **13-line** Haskell implementation, demonstrating that $sodt_{\text{vec}}$ and $sodt_{rec}$ enumerate the same set of trees in different orders. By contrast, classical ODT algorithms are often specified informally and at greater length. For example, Demirović et al. (2022)'s algorithm spans nearly **five pages**, while leaving numerous functions undefined, and addresses the simpler ODT-BF problem.

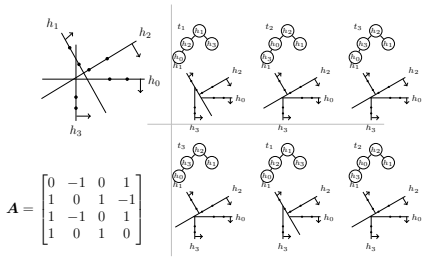


Figure 3: Left: four hyperplanes and corresponding ancestry relation matrix $\boldsymbol{A}$; circles mark defining points and arrows indicate orientations. Right: the three decision trees and induced partitions generated by (He & Little, 2025b)'s algorithm (above). Our algorithm generates the same trees in different orders (below). Both methods prune the 24 possible trees to three.

Since depth- and size-constrained trees have different search spaces, direct comparison of our approach with existing ODT solvers is challenging. Nevertheless, $sodt_{rec}$ is the closest size-constrained analogue of depth-constrained methods because it uses the same branching methods; only the split generator follows from the chosen constraint. We therefore compare $sodt_{rec}$ with these methods and show that it is more efficient in both sequential and parallel settings.

Extensive results reported in Subsection 4.1 and Appendix E.1.1 show that our proposed method is more efficient in both sequential and parallel settings.

3.4 ANCESTRY RELATION MATRICES AND SAFE GEOMETRIC PRUNING

The vectorized search described in the preceding sections implicitly uses an ancestry relation matrix $\boldsymbol{A} \in \{-1, 0, 1\}^{K \times K}$ to guide $updates_{\boldsymbol{A}}$. Its entry $\boldsymbol{A}_{ij}$ records whether rule h_j may occur below rule h_i in the positive branch (1), the negative branch (-1), or neither branch (0). However, we have not yet explained how to generate this matrix, and doing so efficiently is a major computational challenge. Moreover, we show that, when $\boldsymbol{A}$ satisfies a certain geometric property, it can safely prune the decision-tree search space without excluding any decision tree. Thus, this property serves as a pruning certificate. In this section, we formalize the property and show how to incorporate it efficiently into our algorithm.

For two arbitrary rules, the ancestry matrix distinguishes three cases:

1. **Mutual ancestry**: both rules can serve as ancestors of the other, so $\boldsymbol{A}_{ij} \neq 0$ and $\boldsymbol{A}_{ji} \neq 0$.

2. **Asymmetrical ancestry**: exactly one rule can serve as an ancestor of the other, so exactly one of $\boldsymbol{A}_{ij}$ and $\boldsymbol{A}_{ji}$ is nonzero.
3. **No ancestry**: neither rule can serve as an ancestor of the other, so $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji} = 0$.

To formalize them, we first instantiate *Partition transitivity* of the PDT axioms for hyperplane splits:

Let X_ℓ be the set of data points that defines h_ℓ. If h_ℓ is contained in a decision region R, then $X_\ell \subseteq R$. (5)

Thus, mutual ancestry requires $(X_j \subseteq h_i^+ \text{ or } X_j \subseteq h_i^-) \wedge (X_i \subseteq h_j^+ \text{ or } X_i \subseteq h_j^-)$, whereas asymmetrical ancestry requires only one of these containments. If neither containment holds, neither rule can be an ancestor of the other. Figure 8 illustrates these cases for hyperplane splits in $\mathbb{R}^2$.

However, characterizing the no-ancestry case as the complement of the first two cases is neither intuitive nor straightforward to check during computation. Can we instead impose a geometrically clearer condition that guarantees the no ancestry relation? The answer is yes, although the resulting condition is more intricate than it first appears.

Our observations in $\mathbb{R}^2$ suggest that two hyperplanes have no ancestry relation when each has defining points on both sides of the other. We formalize this intuitive crossing pattern below.

Mutual straddling is not a pruning certificate

Definition (Straddling and crossed-hyperplanes (CH)). *Let $X \subset \mathbb{R}^D$ be a finite point set and let $h = \{x : f(x) = 0\}$ be an affine hyperplane with halfspaces h^+ and h^-. The* straddling predicate *is*

$$p_{\text{str}}(X, h) := \big[X \cap h^+ \neq \varnothing\big] \wedge \big[X \cap h^- \neq \varnothing\big].$$

In this case, we say that X straddles *h. The crossed-hyperplane predicate is defined for two hyperplanes h_i, h_j that* mutually straddle *each other:*

$$p_{\text{crs}}(h_i, h_j) := p_{\text{str}}(X_i, h_j) \wedge p_{\text{str}}(X_j, h_i). \tag{6}$$

However, the CH property alone does not guarantee that h_i and h_j have no ancestry relation. In particular, it does not rule out a larger tree in which a third rule places h_i and h_j in different branches. This distinction is illustrated by the counterexample in Figure 7 for $\mathbb{R}^3$.

The limitation arises when mutual straddling is used to infer intersection of the corresponding convex hulls. In $\mathbb{R}^2$, mutual straddling implies that both convex hulls meet $h_i \cap h_j$. For $D \geq 3$, however, they may meet this intersection at different locations and therefore remain disjoint. We then strengthen mutual straddling by requiring the convex hulls of the defining point sets to intersect. As we show next, this condition yields a sound pruning criterion in every dimension.

Theorem 3 (Sound fatal-pair pruning). *Let h_i and h_j be affine hyperplane rules with defining point sets X_i and X_j. If the* fatal-pair *predicate*

$$p_{\text{fatal}}(h_i, h_j) := p_{\text{str}}(X_i, h_j) \wedge p_{\text{str}}(X_j, h_i) \wedge [\text{conv}(X_i) \cap \text{conv}(X_j) \neq \varnothing] \tag{7}$$

holds, where $\text{conv}(X_i)$ denotes the convex hull of X_i, then neither rule can be an ancestor of the other; no affine hyperplane rule can place them in different branches; and no proper decision tree satisfying (5) *can contain both rules.*

The proof is given in Appendix B.3. The result extends to degree-M polynomial rules after applying the monomial feature map $\phi_M : \mathbb{R}^D \to \mathbb{R}^G$ from Theorem 1.

Safe filtering and incremental generation A tree of size K contains K rules. A configuration is safe precisely when it contains no fatal pair. Accordingly, we define the configuration predicate as the conjunction of the negations of all pairwise fatal-pair predicates:

$$p_{\text{safe}}(hs) := \bigwedge_{1 \leq i < j \leq |hs|} \neg p_{\text{fatal}}(h_i, h_j), \tag{8}$$

To replace the $nestedCombs$ procedure in Algorithm 3, we introduce a filtered version that incorporates this predicate:

$$nestedCombsFA(K, G) = filter_{p_{\text{safe}}} \circ map_{calARM} \circ nestedCombs(K, G). \tag{9}$$

Evaluating this filter after generating a candidate requires $O(K^2)$ pairwise fatal-pair tests.

This $O(K^2)$ cost per candidate is inefficient because $nestedCombs(K, G)$ produces $O(N^{KG})$ candidates. Crucially, the predicate in (8) is **prefix-closed**: if a prefix of hs is unsafe, then hs is also unsafe. We can therefore filter configurations incrementally during generation, rather than only after they have been fully constructed. Accordingly, we develop Algorithm 4, which checks the safety of a configuration as it is built. The following result shows that this algorithm has lower complexity than the original $nestedCombs$ procedure in Algorithm 3.

Theorem 4. *The programs $filter_{p_{\text{safe}}}$, map_{calARM}, and $nestedCombs$ can be fused into the single algorithm $nestedCombsFA$, as defined in Algorithm 4, thereby reducing the number of fatal-pair tests per generated configuration from $O\left(K^2\right)$ to $O\left(K\right)$.*

Combining the preceding results, the overall worst-case work complexity of the odt' algorithm is $O\left(K!KN^{KG+1}\right)$ under the assumptions stated in Theorem 13, as proved in Theorem 13.

4 Empirical Results

We evaluate four aspects of the proposed method: (i) the sequential and parallel scalability of $sodt_{\text{vec}}$ relative to $sodt_{\text{rec}}$; (ii) the effectiveness of sound fatal-pair pruning; (iii) Evaluating the robustness of HDT trees over different source of noise: tree sizes, dimensions, sample sizes, and noise on labels/data, each one is evaluated on 50 synthetic datasets; (iv) generalization performance on 30 real-world datasets. All experiments use an Intel i9-14900KF CPU and a single GeForce RTX 4060 Ti GPU. We report the main results here and defer additional results and discussion to Appendix E.

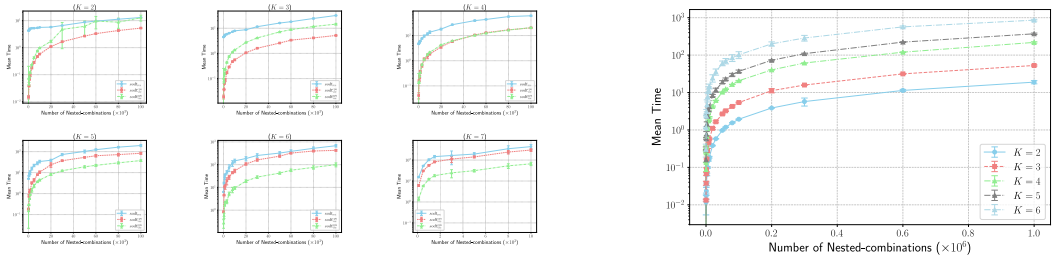


Figure 4: Parallel run times for $sodt_{\text{rec}}$ and $sodt_{\text{vec}}$ across K (left), and log-log run time for $sodt_{\text{vec}}$ up to 1×10^6 nested combinations (right).

4.1 Computational Scalability Comparison

Figure 4 shows that $sodt_{\text{vec}}$ is substantially faster than $sodt_{\text{rec}}$ in parallel. For $K = 2$, $sodt_{\text{vec}}$ explores 0.6×10^6 feasible nested configurations—corresponding to up to 1.2×10^6 candidate trees—in approximately 10 seconds. Appendix E.1 presents the sequential comparison.

4.2 Effectiveness of the Safe Pruning

As shown in Figure 5, the combinatorial complexity after safe filtering (bars) is substantially lower than the theoretical upper bound $O(N^{DK})$ (black lines). Moreover, the complexity curve is inverted U-shaped, indicating that the proportion of infeasible configurations increases with K and does so at an increasing rate. Appendix E.2 presents additional results for other values of D and M.

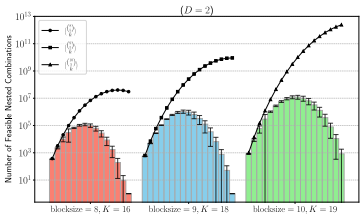


Figure 5: Combinatorial complexity of decision tree search space

4.3 Analysis on Synthetic Datasets

This experimental design builds on the setup of Murthy & Salzberg (1995), extending its ADT experiments to the hyperplane setting.

Table 1 presents selected results from Table 6, illustrating the effect of label noise across methods. Tables 3, 5, and 7 in Appendix E summarize the effects of the remaining factors. As label noise increased, the accuracy of all methods declined, with similar reductions in out-of-sample performance up to 20% noise. Between 20% and 25% noise, however, the accuracy of CART-depth, CART-size, and ConTree decreased by 1.47%, 1.73%, and 1.55%, respectively, compared with only 0.91% for HDT. Moreover, from a noise level of 5% onward, HDT found solutions that *exceeded the ground truth on the training dataset* while maintaining strong out-of-sample performance. These results provide insights that high-training accuracy does not necessarily indicate overfitting, and provide evidence against that optimal methods necessarily overfit.

Noise	Method	Train acc.	Test acc.	Size	$D_{\max}$	D_{avg}	$D_{\exp}$
5%	CART-depth-2	86.29 (4.48)	87.15 (4.89)	2.94 (0.24)	2.00 (0.00)	1.98 (0.08)	1.99 (0.02)
	CART-size-4	90.64 (3.07)	91.01 (3.67)	4.00 (0.00)	3.20 (0.40)	2.50 (0.16)	2.10 (0.24)
	ConTree-2	89.07 (3.45)	89.87 (3.47)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	95.00 (0.20)	97.20 (1.40)	2.00 (0.00)	2.00 (0.00)	1.68 (0.00)	1.62 (0.29)
	Ground truth	95.00 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
15%	CART-depth-2	77.90 (3.99)	85.99 (5.71)	2.98 (0.14)	2.00 (0.00)	1.99 (0.05)	2.00 (0.04)
	CART-size-4	81.80 (3.10)	90.19 (3.67)	4.00 (0.00)	3.22 (0.41)	2.51 (0.17)	2.19 (0.31)
	ConTree-2	80.53 (3.04)	89.10 (3.79)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	85.69 (0.54)	97.10 (1.84)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.63 (0.29)
	Ground truth	84.99 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
25%	CART-depth-2	70.35 (3.22)	84.59 (6.66)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	1.99 (0.09)
	CART-size-4	73.19 (2.54)	87.78 (5.41)	4.00 (0.00)	3.26 (0.44)	2.52 (0.17)	2.05 (0.35)
	ConTree-2	72.26 (2.62)	87.06 (5.16)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	1.99 (0.05)
	HDT	76.37 (0.83)	95.67 (2.93)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.78)
	Ground truth	75.00 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)

Table 1: Selected results from Table 6 for the effect of label noise. Entries are means (standard deviations).

4.4 Analysis on real-world datasets

This paper aims to demonstrate that more expressive tree models can learn the ground truth more accurately. Because HSDTs have intractable combinatorial complexity, they are difficult to optimize within a reasonable time on real-world datasets. We therefore compare HDTs with classical ADTs. With model complexity controlled, HDTs consistently outperform ADTs across a range of real-world and synthetic datasets. In the strongest cases, HDTs improve training accuracy by more than 20% and test accuracy by nearly 30% compared with the optimal ADT algorithm.

Table 2 reports only Part I of the experimental results. Part II is reported in Table 8. Appendix E.4 provides further details on the experimental settings and results for larger tree sizes.

Dataset	N	D	C	CART-depth $d=2$	ConTree $d=2$	CART-size $K=2$	CART-size $K=3$	HDT $K=2$	HDT $K=3$
haberman	283	3	2	75.58/73.68 (1.62/3.14)	76.99/72.98 (0.93/3.06)	74.25/72.63 (1.26/4.52)	75.31/73.68 (1.82/2.94)	79.38/**78.25** (1.23/3.60)	**80.70**/78.24 (1.39/4.87)
BldTrns	502	4	2	75.71/72.08 (1.05/2.02)	77.16/71.09 (0.60/1.70)	75.71/72.08 (1.05/02.02)	76.91/74.06 (0.56/03.09)	79.38/**78.50** (1.16/4.37)	**80.71**/78.25 (1.12/3.86)
sepsis	975	3	2	94.13/93.64 (0.30/1.06)	94.26/93.74 (0.30/0.88)	94.13/93.64 (0.30/1.06)	94.18/93.64 (0.35/1.06)	94.97/**93.85** (0.44/1.95)	**95.39/93.85** (0.43/1.95)
algerian	243	14	2	99.28/98.37 (0.25/1.53)	99.69/95.92 (0.25/2.24)	99.28/98.37 (0.25/1.52)	99.59/98.37 (0.21/1.53)	99.59/97.96 (0.43/1.44)	**100/98.78** (0.00/1.12)
Cryotherapy	89	6	2	94.93/90.00 (1.91/6.48)	94.93/91.11 (1.91/6.67)	91.83/80.00 (2.87/10.30)	94.93/90.00 (1.91/6.48)	98.31/93.33 (0.63/6.33)	**99.16/93.33** (0.77/4.65)
Caesarian	72	5	2	73.33/58.67 (2.05/7.78)	75.09/57.33 (1.31/6.80)	73.33/61.33 (2.05/7.78)	75.79/64.00 (2.05/9.04)	88.42/**85.33** (1.57/5.58)	**91.23**/84.00 (2.15/3.68)
ecoli	336	7	8	80.37/79.12 (1.28/2.35)	81.11/**80.59** (0.80/3.65)	76.04/75.00 (1.38/2.94)	81.11/77.35 (0.80/1.77)	80.73/77.64 (0.97/4.08)	**82.76**/79.12 (1.10/3.81)
GlsId	213	9	6	62.35/62.33 (1.18/5.38)	67.29/**62.79** (1.15/04.88)	61.76/62.33 (1.18/5.38)	66.47/63.72 (1.12/5.22)	71.18/61.86 (2.46/4.53)	**74.25/62.79** (2.29/5.20)

Table 2: (Part I: K in 2, 3) Five-fold cross-validation results. We compare the performance of our HDT model, with K (number of splitting rules) ranging from 2 to 3 and the depth is fixed as 2, against approximate methods:size- and depth-constrained CART algorithms (CART-size and CART-depth), as well as the state-of-the-art optimal ADT algorithm, ConTree (Brita et al., 2025). Results are reported as mean accuracy on the training and test sets in the format *Training Error / Test Error (Standard Deviation: Train / Test)*. The best-performing algorithm in each row is shown in **bold**. Complete results appear in Table 8.

5 Conclusion

We present the first optimal hypersurface decision tree algorithm, with two complementary speed-ups. First, a generic vectorized solver for the $sodt$ problem enables efficient GPU parallelization, outperforms custom dynamic-programming approaches even on a sequential CPU, and applies to any tree in the PDT family. Second, for HDTs and HSDTs, safe fatal-pair pruning is fused directly into candidate generation. Experiments confirm the efficiency of both techniques and show that, under controlled model complexity, our algorithm significantly outperforms classical axis-parallel decision trees on synthetic and more than 30 real-world datasets.

AI USE STATEMENT

In this work, we used generative AI tools for polishing English grammar and spelling, and for improving the clarity of the text. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

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

Gaël Aglin, Siegfried Nijssen, and Pierre Schaus. Pydl8. 5: a library for learning optimal decision trees. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pp. 5222–5224, 2021.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Dimitris Bertsimas and Jack Dunn. Optimal classification trees. *Machine Learning*, 106:1039–1082, 2017.

Dimitris Bertsimas and Jack Dunn. *Machine learning under a modern optimization lens*. Dynamic Ideas LLC Charlestown, MA, 2019.

Richard Bird and Oege De Moor. The algebra of programming. *NATO ASI DPD*, 152:167–203, 1996.

Richard Bird and Jeremy Gibbons. *Algorithm Design with Haskell*. Cambridge University Press, 2020.

Justin Boutilier, Carla Michini, and Zachary Zhou. Optimal multivariate decision trees. *Constraints*, 28(4):549–577, 2023.

L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN 9780412048418. URL https://books.google.co.uk/books?id=JwQx-WOmSyQC.

Cătălin E Brita, Jacobus GM van der Linden, and Emir Demirović. Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 11131–11139, 2025.

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

Jack William Dunn. *Optimal trees for prediction and prescription*. PhD thesis, Massachusetts Institute of Technology, 2018.

Xuhui Fan, Bin Li, and Scott Sisson. The binary space partitioning-tree process. In *International Conference on Artificial Intelligence and Statistics*, pp. 1859–1867. PMLR, 2018.

Oktay Günlük, Jayant Kalagnanam, Minhan Li, Matt Menickelly, and Katya Scheinberg. Optimal decision trees for categorical data via integer programming. *Journal of Global Optimization*, 81: 233–260, 2021.

Xi He. Foundational theory for optimal decision tree problems. i. algorithmic and geometric foundations, 2025. URL https://arxiv.org/abs/2509.11226.

Xi He and Max. A. Little. Combination generators with optimal cache utilization and communication free parallel execution, 2025a. URL https://arxiv.org/abs/2507.03980.

Xi He and Max A. Little. Proper decision trees: An axiomatic framework for solving optimal decision tree problems with arbitrary splitting rules, 2025b. URL https://arxiv.org/abs/2503.01455.

Xi He and Max A Little. An efficient, provably exact algorithm for the 0-1 loss linear classification problem. *The Fourteenth International Conference on Learning Representations (ICLR 2026)*, 2026.

Xi He, Yi Miao, and Max A. Little. DeepICE: The first globally optimal algorithm for empirical risk minimization of two-layer maxout and relu networks. *The Fourteenth International Conference on Learning Representations (ICLR 2026)*, 2026.

Xiyang Hu, Cynthia Rudin, and Margo Seltzer. Optimal sparse decision trees. *Advances in Neural Information Processing Systems*, 32, 2019.

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

Jimmy Lin, Chudi Zhong, Diane Hu, Cynthia Rudin, and Margo Seltzer. Generalized and scalable optimal sparse decision trees. pp. 6150–6160. Proceedings of Machine Learning Research, 2020.

Rahul Mazumder, Xiang Meng, and Haoyue Wang. Quant-bnb: A scalable branch-and-bound method for optimal decision trees with continuous features. In *International Conference on Machine Learning*, pp. 15255–15277. PMLR, 2022.

Sreerama K Murthy and Steven Salzberg. Decision tree induction: How effective is the greedy heuristic? In *KDD*, pp. 222–227, 1995.

Siegfried Nijssen and Elisa Fromont. Mining optimal decision trees from itemset lattices. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 530–539, 2007.

Siegfried Nijssen and Elisa Fromont. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining and Knowledge Discovery*, 21:9–51, 2010.

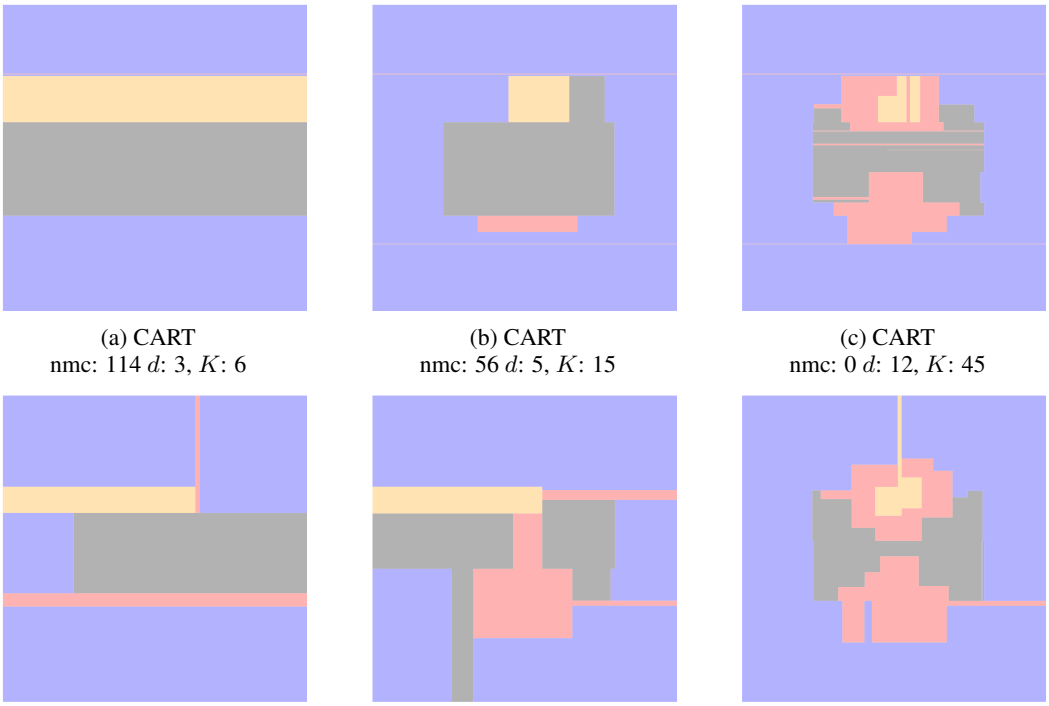
Csaba D Tóth. Binary space partitions: recent developments. *Combinatorial and Computational Geometry*, 52:525–552, 2005.

Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 1625–1632, 2019.

Rui Zhang, Rui Xin, Margo Seltzer, and Cynthia Rudin. Optimal sparse regression trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 11270–11279, 2023.

Haoran Zhu, Pavankumar Murali, Dzung Phan, Lam Nguyen, and Jayant Kalagnanam. A scalable mip-based method for learning optimal multivariate decision trees. *Advances in neural information processing systems*, 33:1771–1781, 2020.

A ADDITIONAL COMPARISON



(a) CART
nmc: 114 d: 3, K: 6

(b) CART
nmc: 56 d: 5, K: 15

(c) CART
nmc: 0 d: 12, K: 45

(d) ConTree
nmc: 69 d: 3, K: 7

(e) ConTree
nmc: 34 d: 4, K: 15

(f) ConTree
nmc: 0 d: 6, K: 48

Figure 6: Same ground truth synthetic dataset as 1. The above three models are axis-parallel decision trees learned by CART and the below three are axis-parallel decision trees learned by ConTree algorithm.

B PROPER DECISION TREE

B.1 DEFINITION OF POLYNOMIAL HYPERSURFACE

A polynomial hypersurface in $\mathbb{R}^D$ is the zero set of a polynomial function $f : \mathbb{R}^D \to \mathbb{R}$, i.e., the set of points $\mathbf{x} = (x_1, x_2, \ldots, x_D)$ such that $f(\mathbf{x}) = 0$. Let $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_D)$ be a multi-index exponent vector, and $\mathbf{x}^\alpha = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \ldots x_D^{\alpha_D}$. The degree of a polynomial is defined as the maximum total degree of its monomial terms, i.e., $|\boldsymbol{\alpha}|$. For example, a hyperplane corresponds to a first-degree polynomial and can be written as $w_1 x_1 + w_2 x_2 \ldots w_D x_D + w_0 = 0$ while a conic section (a quadratic polynomial) can be expressed as $w_1 x_1^2 + w_2 x_1 x_2 + \ldots + w_D x_D^2 + w_0 = 0$.

B.2 AXIOMATIC DEFINITION

1) Each branch (internal) node of the decision tree subdivides the ambient space into two *disjoint* and connected subspaces 2) and the leaves of the decision tree is the intersection of subspaces for all the splitting rules 3) the relationship between splitting rules is transitive, namely if r_j lies in the subspace of r_i and r_i lies in the subspace of r_k, then r_j lies in the subspace of r_k; 4) Notably a distinguishing feature of He (2025)'s axiom is that the left-children of a splitting rule **cannot** be the right-children.

Formally, we have following definitions

Definition 5. Axioms for proper decision trees. *We call a decision tree consists of splitting rules that satisfies the following axioms, a* proper *decision tree:*

1. *Structural constraint one* (Ambient space partition): Each branch node is defined by a single splitting rule $r : \mathcal{R}$, and each splitting rule subdivides the ambient space into two *disjoint* and connected subspaces, r^+ and r^-.

2. *Structural constraint two* (Path intersection in leaves): Each leaf L is defined by the intersection of subspaces $\bigcap_{p \in P_L} r_p^{\pm}$ for all the splitting rules $\{r_p \mid p \in P_L\}$ in the path P_L from the root to leaf L. The connected region (subspace) defined by $\bigcap_{p \in P_L} r_p^{\pm}$ is referred to as the *decision region*.

3. *Structural constraint three* (Partition transitivity): The ancestry relation between any pair of splitting rules $r_i \, (\swarrow \vee \searrow) \, r_j$ is transitive; in other words, if $r_i \, (\swarrow \vee \searrow) \, r_j$ and $r_j \, (\swarrow \vee \searrow) \, r_k$ then $\neg(r_i \swarrow r_j) \wedge \neg(r_i \swarrow r_j)$. Moreover, $\boldsymbol{A}_{ij} = \pm 1$ if r_j can be generated from $r_i^{\pm}$. As a result, any new decision rule r added to a leaf must be generated within its corresponding decision region.

4. *Ancestral constraint* (Uniqueness of the ancestry relation): For any pair of splitting rules r_i and r_j, only one of the following three cases is true: $r_i \swarrow r_j$, $r_i \searrow r_j$, and $r_i \overline{(\swarrow \vee \searrow)} r_j$; additionally, $r_i \overline{(\swarrow \vee \searrow)} r_i$ is always true; in other words, the possible value of $\boldsymbol{A}_{ij} \in \{1, 0, -1\}$ is unique determined for all i, j , and $\boldsymbol{A}_{ii} = 0$ for all i.

There are two important theorems that can be derived based on the PDT axioms.

Theorem 6. *Proper decision tree characterization theorem.* A decision tree consisting of K splitting rules corresponds to a unique K-permutation if and only if it is proper.

Furthermore, K-permutations are among the most extensively studied combinatorial objects in the literature. By characterizing decision trees in terms of these permutations, we obtain the following theorem, which describes how to decompose the size-constrained ODT problem as follows.

Theorem 7. *Simplified optimal decision tree problem.* Given a list of rules $rs : [\mathcal{R}]$ and a size constraint $K : \mathbb{N}$. Let the search space $\mathcal{S}(K, rs)$ of size-K decision trees be defined by the program $genDTKs(K, rs)$. Then there exists a dynamic programming algorithm $sodt$ such that the following equivalence holds:

$$min_E(concatMapL(sodt, kcombs(K, rs))) \subseteq min_E(genDTKs(K, rs)) \tag{10}$$

where the symbol "$\subseteq$" indicates that the solution on the left-hand side is also a solution on the right-hand side. Here, $kcombs(K, rs)$ generates all possible K-combinations of rules from rs. The function $concatMapL$ applies $sodt$ to each combination returned by $kcombs(K, rs)$ and flattens the resulting list of lists into a single list. Finally, the operator min_E is the programmatic definition of argmin, which selects the *first* optimal solution with respect to E from a list of candidates.

See He & Little (2025b) for the proof of Theorem 6 and Theorem 7.

B.3 PROOF OF THEOREM 3

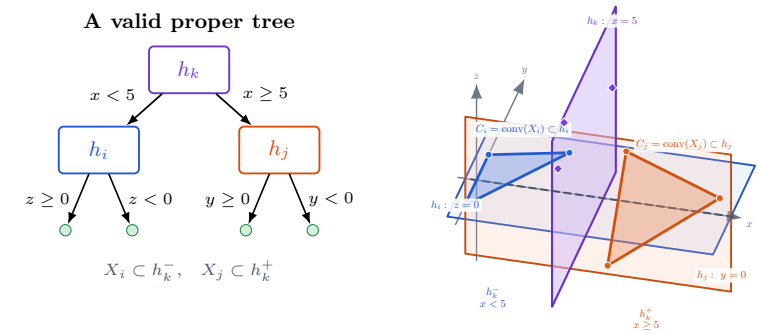


Figure 7: Let $h_i = \{(x, y, z) : z = 0\}$ with $X_i = \{(4, 3, 0), (0, 2, 0), (0, -2, 0)\}$, $h_j = \{(x, y, z) : y = 0\}$ with $X_j = \{(13, 0, 1), (7, 0, -4), (8, 0, 3)\}$, and $h_k = \{(x, y, z) : x = 5\}$ with $X_k = \{(5, 7, 1), (5, -2, 3), (5, -1, 5)\}$. Each set contains three affinely independent points, and the nine points are in general position (no four are coplanar). Since the y-coordinates in X_i are $3, 2, -2$ and the z-coordinates in X_j are $1, -4, 3$, both $p_{\mathrm{str}}(X_i, h_j)$ and $p_{\mathrm{str}}(X_j, h_i)$ hold; hence $p_{\mathrm{crs}}(h_i, h_j)$ is true. Nevertheless, $X_i \subset \{x < 5\} = h_k^-$ and $X_j \subset \{x > 5\} \subset h_k^+$. In other words, the blue hyperplane (h_i) and yellow hyperplanes (h_j) are crossed (mutually straddling), but there exists a third hyperplane (h_k) that can separate them.

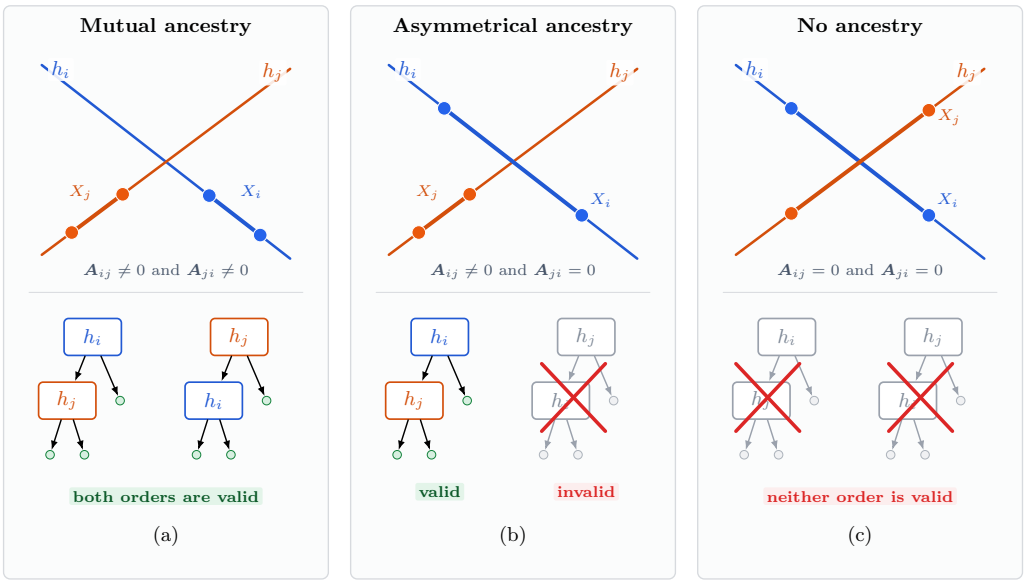


Figure 8: The three pairwise ancestry relations. The upper row shows the two rules and their defining point sets; the lower row shows the corresponding tree structures. (a) In mutual ancestry, both root orders are valid. (b) In asymmetrical ancestry, only one root order is valid. (c) In no ancestry, neither rule can be a direct ancestor of the other. The last case concerns direct ancestry only and does not, without an additional geometric condition, rule out the two rules appearing in separate branches of a larger tree.

Theorem 8 (Sound fatal-pair pruning)**.** *Let h_i and h_j be affine hyperplane rules with defining point sets X_i and X_j. If $p_{\mathrm{fatal}}(h_i, h_j)$ holds, then neither rule can be an ancestor of the other, no affine hyperplane rule can place them in different branches, and no proper decision tree satisfying* (5) *can contain both rules.*

Proof. Write $C_i = \mathrm{conv}(X_i)$ and $C_j = \mathrm{conv}(X_j)$. We prove the result in three steps.

Step 1: exclude an ancestry relation between h_i and h_j. Suppose first that h_i is an ancestor of h_j. The node h_j must then occur in exactly one of the two branches of h_i. By (5), all points defining h_j must belong to the decision region in which h_j is generated. In particular, either

$$X_j \subseteq h_i^+ \qquad \text{or} \qquad X_j \subseteq h_i^-.$$

This is impossible because X_j contains points in both h_i^+ and h_i^-. Therefore h_i cannot be an ancestor of h_j. Interchanging i and j gives the same contradiction, so h_j cannot be an ancestor of h_i either. Thus

$$h_i \not\prec h_j \qquad \text{and} \qquad h_j \not\prec h_i. \tag{3}$$

This step uses only the mutual-straddling assumption.

Step 2: exclude a third rule that separates the two nodes. Suppose, for contradiction, that an affine hyperplane rule h_k places h_i and h_j in different branches. After possibly interchanging the branch labels, let h_i lie in the h_k^+ branch and h_j in the h_k^- branch. Repeated application of (5) along the two ancestor paths gives

$$X_i \subseteq h_k^+, \qquad X_j \subseteq h_k^-. \tag{1}$$

Both halfspaces are convex. Therefore this implies

$$C_i \subseteq h_k^+, \qquad C_j \subseteq h_k^-. \tag{2}$$

For completeness, if $x_1, \ldots, x_m \in h_k^-$, $\lambda_r \geq 0$, and $\sum_r \lambda_r = 1$, then

$$f_k\left(\sum_{r=1}^m \lambda_r x_r\right) = \sum_{r=1}^m \lambda_r f_k(x_r) < 0.$$

so the strict negative halfspace is also closed under convex combinations. Since $h_k^+ \cap h_k^- = \varnothing$, (2) yields $C_i \cap C_j = \varnothing$. This contradicts (7). Consequently, no affine hyperplane rule can place h_i and h_j in different branches.

This step is exactly where the convex-hull-intersection assumption is needed; mutual straddling alone does not rule out such a separator in dimensions $D \geq 3$.

Step 3: exhaust the two possible positions in a rooted tree. Suppose, finally, that a proper rooted binary decision tree contains both h_i and h_j. Any two nodes in a rooted tree fall into exactly one of the following cases:

1. *Comparable case.* One node is an ancestor of the other. This is impossible by (3).
2. *Incomparable case.* Neither node is an ancestor of the other. Let h_k be their lowest common ancestor. Then h_i and h_j lie below different children of h_k. Repeated application of (5) along their paths places X_i and X_j in opposite halfspaces of h_k. Hence h_k is precisely a separating rule of the kind excluded in Step 2.

The two cases exhaust all possible relative positions of two nodes in a rooted tree, and both are impossible. Therefore no proper decision tree satisfying (5) can contain both h_i and h_j. □

Convex hulls in the original input space cannot replace (11), because ϕ_M is nonlinear. For finite defining sets, the lifted intersection is exactly the following linear feasibility problem:

$$\begin{aligned} \sum_r \alpha_r \phi_M(x_{i,r}) &= \sum_s \beta_s \phi_M(x_{j,s}), \\ \alpha_r, \beta_s &\geq 0, \\ \sum_r \alpha_r = 1, &\qquad \sum_s \beta_s = 1. \end{aligned} \tag{11}$$

This provides an implementable and dimension-independent safe test.

C ALGORITHMS

C.1 A NOVEL, PARALLELIZABLE AND GENERIC ALGORITHM FOR SIMPLIFIED OPTIMAL DECISION TREE PROBLEM

C.1.1 RECURSIVE DEFINITION

Building on this idea, the following function exhaustively generate all decision trees in the search space

$$\begin{aligned} &genDTs_{\text{vec}} : \mathcal{D} \times [\mathcal{R}] \to [DTree\,(\mathcal{R}, \mathcal{D})] \\ &genDTs_{\text{vec}}\,(xs, [\,]) = [DL\,(xs)] \\ &genDTs_{\text{vec}}\,(xs, rs) = concat \circ [updates_{\mathbf{A}}\,(r, rs \backslash r, xs) \mid (r, rs \backslash r) \longleftarrow candidates\,(rs)] \end{aligned} \tag{12}$$

where $candidates\,(rs) = [(r, rs \backslash r) \mid r \leftarrow rs]$ generates all possible rules in rs and $rs \backslash r$ denotes eliminating r from list rs, and $updates$ is defined as

$$\begin{aligned} &updates_{\mathbf{A}} : \mathcal{R} \times [\mathcal{R}] \times \mathcal{D} \to [DTree\,(\mathcal{R}, \mathcal{D})] \\ &updates_{\mathbf{A}}\,(r, rs, xs) = catMaybes\,[update_{\boldsymbol{A}}\,(r, t) \mid t \leftarrow genDTs_{\text{vec}}\,(rs, xs)] \end{aligned} \tag{13}$$

which call $genDTs_{\text{vec}}$ recursively, and append a new rule r to every tree t generated by $genDTs_{\text{vec}}\,(rs \backslash r, xs)$ by using $update$ function

$$\begin{aligned} &update_{\boldsymbol{A}} : \mathcal{R} \to DTree\,(\mathcal{R}, \mathcal{D}) \to Maybe\,(DTree\,(\mathcal{R}, \mathcal{D})) \\ &update_{\boldsymbol{A}}\,(r, DL\,(xs)) = Just\,(DN\,(DL\,(xs^{+}), r, DL\,(xs^{-}))) \\ &update_{\boldsymbol{A}}\,(r, DN\,(u, s, v)) = \begin{cases} \begin{cases} DN\,(update_{\boldsymbol{A}}\,(r, u), s, v) & update\,(r, u) \neq Nothing \\ Nothing & \text{otherwise} \end{cases} & \boldsymbol{A}_{sr} = 1 \\ \begin{cases} DN\,(u, s, update_{\boldsymbol{A}}\,(r, v)) & update\,(r, v) \neq Nothing \\ Nothing & \text{otherwise} \end{cases} & \boldsymbol{A}_{sr} = -1 \\ Nothing & \text{otherwise} \end{cases}. \end{aligned} \tag{14}$$

where xs^{+}, xs^{-} are disjoint data sets partitioned by r with respect to xs, and $Maybe\,(\mathcal{A}) = Nothing \mid Just\,(\mathcal{A})$ represents a value that may or may not be present and $catMaybes : [Maybe\,(\mathcal{A})] \to [\mathcal{A}]$ filters out $Nothing$ from a list of $Maybe$ values and extracts the contents of $Just$ values into a plain list. Although the definition of e $update_{\boldsymbol{A}}$ update appears complex, it is applied recursively along the path of the old tree $DN\,(u, s, v)$ to determine whether adding a new splitting rule r would result in a non-proper tree. If the tree remains proper, r is added to the leaf; otherwise, $Nothing$ is returned, indicating a non-proper decision tree. In this way, the algorithm filtered out all non-proper decision tree during the generation process.

Finally, $sodt_{\text{vec}}$ is defined as following

$$sodt_{\text{vec}} = min_{E} \circ genDTs_{\text{vec}}.$$

C.1.2 PROTOTYPE IMPLEMENTATION IN HASKELL

Definition of tree: The binary tree is defined as follows:

```
data Btree a = L | N (Btree a) a (Btree a)
type D = [Int] -- A list of data indexes
type R = Int  -- rule indexes---- D-combination of data
type Asgn = [Int]
```

where `Asgn` is the prediction of hyperplane / hypersurface. To simplify discussion, we assume `D` and `R` are indexes of data points and rules, respectively, thus can be modelled as integers.

Complete definition of gen_{vec}: Our tree generator can be succinctly defined in 13 lines:

```
genBTs_vec :: [R] -> [Btree R]
genBTs_vec [] = [L]
genBTs_vec rs = concat [updates r rs'| (r,rs') <- candidates rs]
 where
  updates x xs = catMaybes [update x t | t <- genBTs_vec xs]
  catMaybes xs = [x | Just x <- xs]
  candidates xs = [(x, xs \\ [x]) | x <- xs]

update :: R -> Btree R -> Maybe (Btree R)
update r L = Just (N L r L)
update r (N u s v)
  | anc_rel_mat !! s !! r == 1  = fmap (\t -> N t s v) (update r u)
  | anc_rel_mat !! s !! r == -1 = fmap (\t -> N u s t) (update r v)
  | otherwise                   = Nothing
```

Input for reproducing Figure 3: We assume there are six data points and four splitting rules:

```
-- Number of data points
n :: Integer
n = 6
asgns :: [Asgn]
asgns = [[1,-1,1,-1,-1,-1],[1,1,-1,1,1,-1],[1,1,1,-1,1,-1],[1,1,-1,-1,1,-1]]
-- Ancestry relation matrix
anc_rel_mat = [[0,-1, 0,1],[1,0,1,-1],[1,-1,0,1],[1,0,1,0]]
```

Filter out duplicates: Note that the gen_{vec} will produce duplicate trees, which is not a problem for the purpose of illustration. This can be solved easily by incorporating following filter in the recursive execution of gen_{vec} without affecting the correctness and asymptotic complexity of the algorithm:

```
genBTs_vec_filt :: [R] -> [Btree R]
genBTs_vec_filt [] = [L]
genBTs_vec_filt rs = filt_dup_t $ concat [updates r rs'| (r,rs') <- candidates rs]
 where
  updates x xs = catMaybes [update x t | t <- genBTs_vec_filt xs]
  catMaybes xs = [x | Just x <- xs]
  candidates xs = [(x, xs \\ [x]) | x <- xs]

traversal :: Btree a -> [a]
traversal tree = go [tree]
  where
    go [] = []
    go (L : rest) = go rest
    go (N u x v : rest) = x : go (rest ++ [u, v])

nubSnd :: Eq b=> [(a, b)] -> [(a, b)]
nubSnd = nubBy (\(_, x) (_, y) -> x == y)

filt_dup_t :: [Btree R] -> [Btree R]
filt_dup_t [t] = [t]
filt_dup_t ts = map fst $ nubSnd $ zip ts (map traversal ts)
```

Classical data-dependent recursive definition: The generator for data dependent recursive $sodt_{\text{rec}}$ is given as follows:

```
genBTs :: [R] -> [Btree R]
genBTs [] = [L]
genBTs [a] = [N L a L]
genBTs hs = [N u r v | (hs_pos, r, hs_neg) <- splits, u <- genBTs hs_pos, v <- gen
 where splits = [split r hs | r <- feasible_split hs]
```

```
feasible_split :: [R] -> [R]
feasible_split [a] = [a]
feasible_split hs = [ hi | hi <- hs, all (ind_mat hi) (filter (/= hi) hs)]
  where ind_mat hi hj = ((anc_rel_mat!!hi)!!hj)/=0

split :: R -> [R] -> ([R], R, [R])
split r [hj] = ([], r, [])
split r [hj,hk]
 |r == hj = if (anc_rel_mat!!r)!!hk == 1 then ([hk], r, []) else ([], r, [hk])
 |r == hk = if (anc_rel_mat!!r)!!hj == 1 then ([hj], r, []) else ([], r, [hj])
split r hs = (hs_pos, r, hs_neg)
 where
  hs_pos = [hj | hj<-hs, (anc_rel_mat!!r)!!hj == 1]
  hs_neg = [hj | hj<-hs, (anc_rel_mat!!r)!!hj == -1]
```

C.1.3 IMPERATIVE DEFINITION

The imperative pseudocode for $sodt_{\text{vec}}$ is presented in Algorithm 1.

Algorithm 1 $sodt_{\text{vec}}$: The imperative definition of the vectorized simplified decision tree problem

Input: $rs : [\mathcal{R}]$: A size K rule list $rs = [r_1, r_2, \ldots, r_K]$; $xs : [\mathbb{R}^D]$: input data list $xs = [x_1, x_2, \ldots, x_N]$, where $x_n \in \mathbb{R}^D$; $\boldsymbol{A} : K \times K$: The ancestry relation matrix, encode the ancestry relation between rules in rs.
Output: $t : DTree\left(\mathcal{R}, \mathbb{R}^D\right)$—The optimal decision tree of size K (K internal node) with respect to rs and xs.

1. $ts = [DN\left(DL\left(xs^+\right), r_i, DL\left(xs^-\right)\right) \mid r_i \in rs]$; // create one candidate tree for each rule as the root
2. for $l = 2, \ldots, K$ do:
3. $\quad ts_{new} = [\,]$;
4. $\quad$ for $t \in ts$ do:
5. $\quad\quad$ for $r_j \in rs \setminus rules(t)$ do: // rules that have not yet been used in t
6. $\quad\quad\quad t' = update_{\boldsymbol{A}}\left(r_j, t\right)$; // update the tree with new rule r_j
7. $\quad\quad\quad$ if $t' \neq Nothing$ do:
8. $\quad\quad\quad\quad$ append t' to ts_{new}; // add the new tree to the new candidate tree list
9. $\quad ts = unique\left(ts_{new}\right)$;
10. **return** $min_E\left(ts\right)$

C.1.4 COMPLEXITY ANALYSIS

The worst-case work complexity of $sodt_{\text{vec}}$ is given by the following theorem.

Theorem 9. *Given a list rs of K rules and a dataset xs of size N, assume that inserting a rule into a partial tree with at most $k-1$ internal nodes costs $O(N+k)$ and that evaluating the objective on a complete size-K tree costs $O(NK)$. Then $sodt_{vec}(xs, rs)$ has worst-case work complexity*

$$O\left(K!\left(KN + K^2\right)\right).$$

In the usual regime $N \geq K$, this simplifies to $O\left(K!KN\right)$.

Proof. Let F_k be the maximum number of candidate trees generated from a list of k rules. Each execution path selects every rule at most once and therefore corresponds to a permutation of the input rules. Invalid insertions are discarded and a valid insertion produces at most one tree, so $F_k \leq k!$.

Let T_k denote the running time of $genDTs_{\text{vec}}$ on k rules. For each of the k choices of the newly inserted rule, the algorithm recursively generates trees from the remaining $k-1$ rules and applies $update_{\boldsymbol{A}}$ to every generated tree. Consequently,

$$T_k \le kT_{k-1} + kF_{k-1}O(N+k) \le kT_{k-1} + k!O(N+k).$$

Dividing by $k!$ and expanding the recurrence gives

$$\frac{T_k}{k!} \le \frac{T_{k-1}}{(k-1)!} + O(N+k), \qquad T_K = O\left(K! \sum_{k=1}^{K} (N+k)\right) = O\left(K!\left(KN + K^2\right)\right).$$

At most $K!$ complete candidate trees are produced. Directly evaluating the objective on all of them adds $O(K!NK)$ work, which is absorbed by the preceding bound. Hence $sodt_{\text{vec}}$ has worst-case work complexity $O\left(K!(KN+K^2)\right)$. When $N \ge K$, this reduces to $O(K!KN)$. □

He & Little (2025b)'s $sodt_{\text{rec}}$ has a complexity of $O\left(K!N\right)$. The two algorithms therefore share factorial dependence on K, but $sodt_{\text{vec}}$ may perform up to an additional polynomial amount of sequential work. Its practical advantage comes from regular memory access and batched parallel execution rather than from a smaller worst-case work bound.

C.2 A NOVEL FATAL-PAIR-FREE NESTED-COMBINATION GENERATOR

C.2.1 NESTED-COMBINATION GENERATOR PROPOSED BY HE AND LITTLE

The nested-combination generator is previously explored by He & Little (2025a) in functional programming setting, we present the imperative version directly here.

Algorithm 2 Nested Combination Generator ($nestedCombs(K, G, xs)$)

Input: xs: data list, length N; K: outer combination size; G: inner-combination size
Output: Array of (K, G)-nested-combinations
$css \leftarrow [[\,], []^G]$ ▷ Initialize combinations
$ncss \leftarrow [[\,], []^K]$ ▷ Initialize nested-combinations
$asgn^+, asgn^- \leftarrow \emptyset\left(\binom{N}{G}, N\right)$ ▷ Empty $\binom{N}{G} \times N$ matrices
for $n \in range(0, N)$ **do**
  for $j \in min(G, n+1), ..., 1$ **do**
    $updates \leftarrow reverse(map(\cup\phi_M(xs)[n], css[j-1]))$
    $css[j] \leftarrow css[j] \cup updates$ ▷ Update combinations
    $asgn^+, asgn^- \leftarrow genModels(css[G], asgn^+, asgn^-)$ ▷ Generate predictions
  end for
  $C_1 \leftarrow \binom{n}{G}, C_2 \leftarrow \binom{n+1}{G}$
  for $i \in range(C_1, C_2)$ **do**
    for $k \in min(K, i+1), ..., 1$ **do**
      $ncss[k] \leftarrow map(\cup[i], ncss[k-1]) \cup ncss[k]$ ▷ Update nested combinations
    end for
  end for
end for
return $ncss[K]$

C.2.2 INCREMENTAL ANCESTRY RELATION MATRIX GENERATOR

As discussed previously, $nestedCombsFA$ is defined by

$$nestedCombsFA\,(K, G) = filter_{p_{\text{safe}}} \circ map_{\,calARM} \circ nestedCombs\,(K, G)\,. \tag{15}$$

The ancestry relation matrix provides a necessary condition for a fatal pair, but it is not by itself a pruning certificate.

Lemma 10. Let $hs_K = [h_1, h_2, \ldots, h_K]$ have ancestry relation matrix $\boldsymbol{A}$. If h_i and h_j form a fatal pair, then $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji} = 0$. The converse does not generally hold.

Proof. A fatal pair mutually straddles, so neither rule can be an ancestor of the other; hence $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji} = 0$. However, zero entries record only the absence of direct ancestry. As shown in Figure 7, mutually straddling rules whose defining convex hulls are disjoint may still occur in different branches of a larger tree. The convex-hull-intersection test in p_{fatal} is therefore also required. □

We construct $nestedCombsFA$ in two steps. First, we update the ancestry relation matrix incrementally. Second, whenever a new rule is added, we test it against the rules already in the prefix using the full fatal-pair predicate, including convex-hull intersection in the lifted feature space.

C.2.3 Incremental update of the ancestry relation matrix

This part we examine how to fuse map_{calARM} with $nestedCombs$, since $nestedCombs$ is an incremental update process, it add one rule to each partial rule list, we need to figure out how to update rule incrementally along the generation process of $nestedCombs$.

Since every rule combination is a nested combination, we create type NC (short for "nested combinations") as a type synonyms for $[\mathcal{R}]$, i.e., $rs : NC$ or $rs : [\mathcal{R}]$. A size K rule combination using type $rs : NC$. Since the ancestry relation matrix $\boldsymbol{A}$ for a K-length rule combination rs_K is a $K \times K$ square matrix. Both $\boldsymbol{A}$ and rs_K are integer-valued and can be stored compactly by stacking rs_K above $\boldsymbol{A}$. That is, a *configuration* $ncr : NCR$ is defined as

$$ncr = \begin{bmatrix} rs_K \\ \boldsymbol{A} \end{bmatrix} \tag{16}$$

Thus, a collection of M configurations can be stored as a tensor of size $M \times (K+1) \times K$.

Thus $nestedCombs\,(K, G) : \left[\mathbb{R}^D\right] \to [NC]$, and $map_{calARM} : [NC] \to [NCR]$, the map is a high-order function which applies its parameter function $calARM : NC \to NCR$ to each $rs : NC$ generated by $nestedCombs\,(K, G)$. A critical observation is that $calARM$ can be updated incrementally.

Specifically, consider a partial configuration ncr_{K-1} consisting of a list of rules rs_{K-1} and a $(K-1) \times (K-1)$ ancestry relation matrix $\boldsymbol{A}'$. We can extend ncr_{K-1} to a complete configuration ncr_K as follows: first, appending a new rule r to rs_{K-1}; second, compute the ancestry relation of r with each rule in rs_{K-1}, and vice versa. This yields the following incremental (sequentially recursive) program $calARM$ for constructing ncr_K

$$\begin{aligned} calARM\,([r]) &= \begin{bmatrix} r \\ 0 \end{bmatrix} \\ calARM\,(r : rs) &= update_{\text{arMat}}\,(r, calARM\,(rs)) \end{aligned}$$

where $\begin{bmatrix} r \\ 0 \end{bmatrix}$ represents a 2×1 matrix. Let $calARM\,(rs) = \begin{bmatrix} rs_k \\ \boldsymbol{K}' \end{bmatrix}$, the update function $update_{\text{arMat}} : NCR \to NCR$ is defined as

$$update_{\text{arMat}}\left(r_j, \begin{bmatrix} rs_k \\ \boldsymbol{A}' \end{bmatrix}\right) = \begin{bmatrix} r_0 & r_1 & \cdots & r_{k-1} & r_j \\ \boldsymbol{A}'_{0,0} & \boldsymbol{A}'_{01} & \cdots & \boldsymbol{A}'_{0,k-1} & AR_{0,j} \\ \boldsymbol{A}'_{1,0} & \boldsymbol{A}'_{1,1} & \cdots & \boldsymbol{A}'_{1,k-1} & AR_{1,j} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \boldsymbol{A}'_{k-1,0} & \boldsymbol{A}'_{k-1,1} & \cdots & \boldsymbol{A}'_{k-1,k-1} & A_{k-1,j} \\ A_{j,0} & A_{j,1} & \cdots & A_{j,k-1} & 0 \end{bmatrix}, \tag{17}$$

which extends a $(k+1) \times k$ matrix to a $(k+2) \times (k+1)$ matrix. The updated matrix consists of:

- $\boldsymbol{A}'$: a $k \times k$ ancestry relation matrix from the previous step,
- rs_k: a $k \times 1$ vector of the indices of the k splitting rules,

- $AR_{i,j}$: the ancestry relation between rule where r_i and r_j, where

$$A_{i,j} = \begin{cases} 1 & \text{if } r_i \text{ can occur in the positive branch below } r_j, \\ -1 & \text{if } r_i \text{ can occur in the negative branch below } r_j, \\ 0 & \text{if } r_i \text{ otherwise.} \end{cases}$$

The incremental nature of the $calARM$ is crucial, as it allows the ancestry matrix to be updated on-the-fly for each rule combination during the recursive generation process of $nestedCombs$. This fusion alone does not reduce the asymptotic cost, but it allows the safe predicate p_{safe} to be evaluated incrementally in the next subsection.

The single step update function $update_{\text{arMat}}$ cab be easily generalized to apply to multiple configurations in a single step, the vectorized version is given in (3). Incorporating this function inside (2) in the loop after $ncss\,[k]$ (line 14) can generate all possible rule combinations with associated ancestry relation matrix associates with it, we will provide the full algorithm after explained the technique introduced next

C.2.4 Prefix-closed filtering for nested combinations

After updating the ancestry relation matrix, we eliminate configurations that contain a fatal pair. A pair of zero ancestry entries is a useful necessary-condition screen, but it is not sufficient for pruning: the defining point sets must also have intersecting convex hulls, as required by p_{fatal} in (7). A post-hoc evaluation of p_{safe} checks $O(K^2)$ rule pairs after the full configuration has already been generated. If one fatal-pair test costs T_{fatal}, this requires $O(K^2 T_{\text{fatal}})$ time per configuration.

To resolve this, we introduce a prefix-closed filtering process. It allows the post-hoc filtering to be *fused directly into the incremental generation process* of $nestedCombs\,(K, G)$, enabling fatal pairs to be identified before partial configurations are extended to full ones. This provides a safe incremental nested-combination generator. Specifically, when we have a generator function, such as $nestedCombs\,(K, G)$, defined sequentially consume one data item in each recursive process, we have the following filter fusion theorem applies.

Theorem 11. *Filter fusion theorem.* Let a sequential generator gen be defined recursively as:

$$\begin{aligned} gen\,([\,]) &= alg_1\,([\,]) \\ gen\,(x : xs) &= alg_2\,(x, gen\,(xs))\,, \end{aligned}$$

and consider a post-hoc filtering process. Then $filtgen_q = filter_p \circ gen$ can be *fused into a single program* defined as:

$$\begin{aligned} filtgen_q\,([\,]) &= filter_q\,(alg_1\,([\,])) \\ filtgen_q\,(x : xs) &= filter_q\left(alg_2\left(x, filtgen_q\,(xs)\right)\right), \end{aligned}$$

provided that the fusion condition holds:

$$filter_p\,(alg_2\,(x, gen\,(xs))) = filter_q\left(alg_2\left(x, filter_p\,(gen\,(xs))\right)\right) \tag{18}$$

In particular, if $alg_2 : \mathcal{A} \times [[\mathcal{A}]] \to [[\mathcal{A}]]$ is defined as an extension operation that prepend $a : \mathcal{A}$ to $as : [\mathcal{A}]$ for all as in $ass : [[\mathcal{A}]]$, then proving the fusion condition (18) is equivalent to proving the *prefix-closed property*:

$$p\,(a : as) = q\,(a : as) \wedge p\,(as) \tag{19}$$

Proof. We prove the fusion theorem by following reasoning

$$\begin{aligned} & filtgen_q \\ =& filter_p \circ gen \\ =& filter_p\,(alg_2\,(x, gen\,(xs))) \\ =& \{\text{fusion condition (18)}\} \\ & filter_q\left(alg_2\left(x, filter_p\,(gen\,(xs))\right)\right) \\ =& \{\text{definition of } filtgen_q\} \\ & filter_q\left(alg_2\left(x, filtgen_q\,(xs)\right)\right) \end{aligned}$$

Algorithm 3 Safe Fatal-Pair Filtering and Ancestry Update ($updates_{\text{safe}}$)

Input: r_j: index of the new rule; $ncrs$: partial safe configurations; $css[G]$: lifted defining point sets; $asgn^{\pm}$: rule predictions
Output: Updated fatal-pair-free configurations $ncrs'$
$ncrs' \leftarrow [\,]$ ▷ Initialize empty list
for $ncr \in ncrs$ **do**
  $safe \leftarrow$ True
  $X_j \leftarrow css[G][r_j]$
  for $r_i \in ncr[0]$ **do**
    $X_i \leftarrow css[G][r_i]$
    $crossed \leftarrow p_{\text{str}}(X_i, h_j) \wedge p_{\text{str}}(X_j, h_i)$
    if $crossed \wedge convIntersects(X_i, X_j)$ **then**
      $safe \leftarrow$ False ▷ $p_{\text{fatal}}(h_i, h_j)$ holds
      break
    end if
  end for
  if $safe$ **then**
    $ncr' \leftarrow update_{arMat}(r_j, ncr, asgn^{\pm})$
    $ncrs' \leftarrow ncrs' + [ncr']$
  end if
end for
return $ncrs'$

The equivalence between filter fusion condition (18) and prefix-closed property (19) is straight forwards. A configuration $a : as$ survived in filter_p will also survived in $\mathit{filter}_q\left(alg_2\left(a, \mathit{filter}_p\left([as]\right)\right)\right)$ because $p\left(a : as\right) = q\left(a : as\right) \wedge p\left(as\right)$. □

The reason the prefix-closed property (19) introduces an additional predicate q is that, if we already know $p\left(as\right)$ holds, it is often more efficient to evaluate $q\left(a : as\right) \wedge p\left(as\right)$ rather than directly computing $p\left(a : as\right)$. This explains why the fused generator $\mathit{filtgen}_q$ is more efficient than the post-hoc approach $\mathit{filter}_p \circ gen$. For example, in the classical *eight queens problem*, p checks that no queen attacks any other, while the auxiliary predicate q only verifies that the newly added queen does not attack the others.

Similarly, for our problem, since $nestedCombs\left(K, D\right)$ is also defined sequentially, we can establish a prefix-closed property for the ancestry relation matrix, analogous to the auxiliary check in the eight queens problem. The correctness of the prefix-closed condition for fatal-pair filtering is given by the following fact.

Fact 12. *Prefix-closed property for fatal pairs.* Assume that hs_K is safe, i.e., $p_{\text{safe}}(hs_K) = \text{True}$. When a new rule h is added, the following identity holds:

$$p_{\text{safe}}\left(h : hs_K\right) = q_{\text{fatal}}\left(h : hs_K\right) \wedge p_{\text{safe}}\left(hs_K\right),$$

where

$$q_{\text{fatal}}\left(h : hs_K\right) = \neg\exists h_i \in hs_K : p_{\text{fatal}}(h, h_i) = \text{True}.$$

Thus, extending a safe prefix requires testing only the new rule against the rules already in that prefix.

Fact 12 reduces the number of pairwise tests from $O(K^2)$ to $O(K)$ per extension, and hence the filtering cost from $O(K^2 T_{\text{fatal}})$ to $O(K T_{\text{fatal}})$.

Here, $convIntersects(X_i, X_j)$ solves the linear feasibility problem in (11). For polynomial rules, X_i and X_j contain the lifted points $\phi_M(x)$; testing convex-hull intersection in the original input space is not sufficient.

We can now incorporate $update_{\text{arMat}}$ along with the safe filtering process for eliminating fatal pairs directly into the definition of $nestedCombs\left(K, G\right)$. For simplicity, we define a batched routine $updates_{\text{safe}}$ that operates on a list of configurations $ncrs$, as implemented in Algorithm 3.

With these components, we can construct the incremental, fatal-pair-free ancestry relation matrix generator $nestedCombsFA$. This is achieved by introducing a single additional line of code after line 13 of Algorithm 2 using Algorithm 3 to update all safe nested combinations while discarding configurations containing a certified fatal pair. The complete pseudo-code is provided in Appendix C.2.2, Algorithm 4.

Algorithm 4 Nested Combination Generator with Ancestry ($nestedCombsFA(K, G, xs)$)

Input: xs: data list, length N; K: outer combination size; G: inner-combination size
Output: Array of (K, G)-nested-combinations
$css \leftarrow [[\,], []^G]$ ▷ Initialize combinations
$ncss \leftarrow [[\,], []^K]$ ▷ Initialize nested-combinations
$asgn^+, asgn^- \leftarrow \emptyset\left(\binom{N}{G}, N\right)$ ▷ Empty $\binom{N}{G} \times N$ matrices
for $n \in range(0, N)$ **do**
  for $j \in min(G, n+1), ..., 1$ **do**
    $updates \leftarrow reverse(map(\cup\phi_M(xs)[n], css[j-1]))$
    $css[j] \leftarrow css[j] \cup updates$ ▷ Update combinations
    $asgn^+, asgn^- \leftarrow genModels(css[G], asgn^+, asgn^-)$ ▷ Generate predictions
  end for
  $C_1 \leftarrow \binom{n}{G}, C_2 \leftarrow \binom{n+1}{G}$
  for $i \in range(C_1, C_2)$ **do**
    for $k \in min(K, i+1), ..., 1$ **do**
      $ncss[k] \leftarrow updates_{\text{safe}}(i, ncss[k-1], css[G], asgn^+, asgn^-) \cup ncss[k]$ ▷ Safe filtering and ancestry update, Alg. 3
    end for
  end for
end for
return $ncss[K]$

C.3 Complexity of the odt' algorithm

Putting the components together, $odt'(xs) = min_E \circ concatMapL(sodt_{\text{vec}}(xs)) \circ nestedCombsFA(xs)$. Its worst-case work complexity is given by the following theorem.

Theorem 13. *Let T_{fatal} be the cost of one fatal-pair feasibility test. The $odt'(xs)$ algorithm has worst-case work complexity*

$$O\left(N^{KG+1} + KT_{\text{fatal}}N^{KG} + K!\left(KN + K^2\right)N^{KG}\right).$$

If $N \geq K$ and the fixed-dimensional fatal-pair test is treated as constant cost, this simplifies to $O\left(K!KN^{KG+1}\right)$.

Proof. Since $odt'(xs) = min_E \circ concatMapL(sodt_{\text{vec}}(xs)) \circ nestedCombsFA(xs)$, the complexity of odt' is the sum of the costs of the composed components and is therefore dominated by the largest term. min_E is linear in the size of the output produced by $concatMapL(sodt_{\text{vec}}(xs))$, thus the complexity odt' is dominated by either $concatMapL(sodt_{\text{vec}}(xs))$ or $nestedCombsFA(xs)$.

As He & Little (2025a) analysed, $nestedCombs(xs)$ produces $O\left(N^{KG}\right)$ configurations in $O\left(N^{KG+1}\right)$ time where K is the number of rules in each configuration, G is the embedded dimension of hypersurface and N is the size of the input xs. Compared with $nestedCombs(xs)$, the $nestedCombsFA$ additionally checks the new rule against at most $K-1$ existing rules. If one fatal-pair test costs T_{fatal}, the additional filtering cost is $O\left(KT_{\text{fatal}}N^{KG}\right)$. When K, G, and the cost of the fixed-dimensional feasibility test are treated as constants, this does not change the stated exponent in N.

The $concatMapL(sodt_{\text{vec}}(xs))$ applies $sodt_{\text{vec}}$ to each of at most $O\left(N^{KG}\right)$ generated configurations. By Theorem 9, its total work is therefore

$$O\left(K!\left(KN + K^2\right)N^{KG}\right).$$

Adding rule-combination generation and fatal-pair filtering yields

$$O\left(N^{KG+1} + KT_{\text{fatal}}N^{KG} + K!\left(KN + K^2\right)N^{KG}\right).$$

When $N \geq K$ and $T_{\text{fatal}} = O(1)$ under a fixed-dimensional cost model, the last term dominates, giving $O\left(K!KN^{KG+1}\right)$. □

D HEURISTIC METHODS

The use of heuristics is pervasive in the study of optimal algorithms. Representative examples include imposing a time limit with random initialization (Dunn, 2018), employing depth-first search with a time limit (Hu et al., 2019; Lin et al., 2020), or using *binarization* for continuous data Brita et al. (2025). Although some BnB algorithms assert that the use of depth-first search with a time limit does not preclude finding the optimal solution if the algorithm is allowed to continue running, this claim is largely vacuous in practice: without a feasible bound on running time, identifying the optimal solution may still require exponential time in the worst case. A similar phenomenon arises in the study of algorithms based on MIP solvers Bertsimas & Dunn (2017; 2019). Although MIP-based methods guarantee attainment of the optimal solution upon full termination, their running times are often unpredictable and impractical for most problems. Consequently, studies employing MIP solvers typically impose time limits and aim to obtain near-optimal solutions within the allotted time, which can likewise be regarded as a heuristic.

It should be emphasized that *the role of these heuristics in the study of optimal algorithms is not to demonstrate their superiority—unless supported by rigorous performance guarantees, which would constitute a distinct line of research—but rather to provide rapid, plausible solutions, thereby illustrating that higher training accuracy under controlled model complexity does not necessarily result in overfitting*. Consequently, most MIP-based algorithms, as well as BnB algorithms that incorporate depth-first search strategies, do not provide guarantees on the quality of the solution obtained within a given execution time.

Therefore, the establishment of rigorous approximation bounds constitutes a separate line of research concerned with the design of approximate algorithms endowed with provable guarantees. By contrast, heuristics in the context of exact algorithms primarily serve to demonstrate the practical effectiveness of exact solutions and to motivate further investigation into scaling exact methods to larger datasets.

The HDT problem exhibits formidable combinatorics. Even for a modest dataset with $N = 100$, $D = 3$, $K = 3$, there are $\left(\begin{pmatrix} 100 \\ 3 \end{pmatrix}\right) \times 3! \approx 4 \times 10^{15}$ possible decision trees in the worst case. We emphasize that **this is not a deficiency of our algorithm but is inherent to the problem itself**, as even its simplest instance—the linear classification problem (a tree with a single internal node)—is NP-hard. Consequently, solving this problem exactly is currently intractable for our algorithms. In this section, we develop two simple yet effective heuristics for addressing the HDT problem.

We do not claim novelty for these two heuristics, nor do we seek to emphasize their intrinsic importance; rather, they are employed as practical tools to provide insight into the behavior of solutions when they are exact or nearly exact. In principle, any other effective heuristics could be adopted for the same purpose. However, without the geometric and algorithmic insights developed in this paper, the construction of such heuristics would not be possible.

D.1 CORESET SELECTION METHOD

The first method, as reported by He et al. (2026), has demonstrated superior performance for empirical risk minimization in two-layer neural networks. The method is based on the following idea: instead of computing the exact solution across the entire dataset—which is computationally infeasible for large K and D—the coreset method identifies the exact solution for the most representative subsets.

The coreset method functions as a "layer-by-layer" data filtering process, in each loop, we run the $hodt$ algorithm for a subset of the dataset, and only keep the L best solutions with respect

Algorithm 5 Coreset Selection ($hodtCoreset(K, M, xs, BS, R, L, B_{\max}, c)$)

Input: Coreset parameters: BS: block size; R: shuffle iterations; L: max-heap size; $B_{\max}$: max input size; $c \in (0, 1]$: shrinking factor
  $hodt$ parameters: K: splitting rules; M: polynomial degree; xs: data list
Output: Max-heap with top L configurations and data blocks
$\mathcal{C} \leftarrow xs$ ▷ Initialize coreset
while $|\mathcal{C}| \geq B_{\max}$ **do**
  $\mathcal{C}_B \leftarrow \{C_1, C_2, \ldots, C_{\lceil |\mathcal{C}|/BS \rceil}\}$ ▷ Reshuffle and divide into blocks
  $\mathcal{H}_L \leftarrow \emptyset_L$ ▷ Initialize size L max-heap
  for $r \in range(1, R)$ **do**
    for $C \in \mathcal{C}_B$ **do**
      $cnfg \leftarrow hodt(K, M, C)$ ▷ Compute configuration
      $\mathcal{H}_L.push(cnfg, C)$ ▷ Push to heap
    end for
  end for
  $\mathcal{C} \leftarrow unique(\mathcal{H}_L)$ ▷ Merge blocks, remove duplicates
  $L \leftarrow L \times c$ ▷ Shrink heap size
end while
$cnfg \leftarrow hodt(K, M, \mathcal{C})$ ▷ Final refinement
$\mathcal{H}_L.push(cnfg, \mathcal{C})$
return $\mathcal{H}_L$

Algorithm 6 $sodt$ with selected hyperplanes ($sodtWSH(K, M, \alpha, xs)$)

Input: K: splitting rules; M: polynomial degree; α: duplicate threshold; xs: data list
Output: Best trees constructed from the selected candidate hyperplanes
$t_{best} \leftarrow \emptyset(K, 1)$ ▷ Initialize vector
$\mathcal{H}_C \leftarrow hodtCoreset(1, M, xs, BS, R, L, B_{\max}, c)$ ▷ Select candidate hypersurfaces
$G \leftarrow \binom{M+D}{D} - 1$
$ncss \leftarrow [[\,], []^k]$ ▷ Initialize nested-combinations
for $n \in range(0, N)$ **do**
  for $k \in reverse(range(K, n+1))$ **do**
    $ncss' \leftarrow filter_{q(\alpha)}(map(\cup[n], ncss[k-1]))$ ▷ Filter duplicates below α
    $t_{best}[k] \leftarrow \min_{E_{0-1}}(mapL(sodt_{xs}, ncss'))$ ▷ Compute best configuration
    $ncss[k] \leftarrow ncss[k] \cup reverse(ncss')$ ▷ Update combinations
    $ncss[K] \leftarrow [\,]$ ▷ Clear size K combinations
  end for
end for
return t_{best}

to the whole dataset. Since better configurations tend to have higher training accuracy, they are more likely to "survive" during the selection process. By recursively reducing the data size until the remaining subset can be processed by running the complete optimal algorithm. The algorithm process is detailed in Algorithm 5.

D.2 *sodt* WITH SELECTED HYPERPLANES

Although the coreset selection method provides plausible solutions for low-dimensional datasets, an obvious limitation arises for high-dimensional datasets: the number of possible splitting rules is too large, even when the dataset is partitioned into small blocks. Moreover, defining a decision tree with K splitting rules requires at least $K \times D$ distinct data points. Running $hodt$ for $K \times D$ points becomes intractable for most high-dimensional datasets. For example, when $K = 2$, and $D = 20$, the block size is 2×20 and the number of possible hyperplanes is $\binom{40}{25} \approx 1 \times 10^{11}$, which is prohibitively large. In practice, our current implementation can efficiently process at most $D + 2$ or $D + 3$ points when $D \geq 20$, far below the ideal block size of $K \times D$.

To address this limitation, we leverage a key advantage of the size-constrained $hodt$ algorithm—its ability to decompose the original difficult ODT problem into many smaller subproblems, each of which can be solved efficiently using $sodt$. Solving the full ODT problem directly is computationally prohibitive due to the enormous combinatorial space. Instead, we focus on finding a set of "good candidate" hyperplanes, and then construct decision trees from these candidates rather than exploring the entire search space.

This insight motivated the development of a new heuristic, $sodtWSH$, (short for "$sodt$ with selected hyperplanes"), described in Algorithm 6. The algorithm identifies candidate hyperplanes with relatively *low training loss* and constructs optimal decision trees based on these hyperplanes. Rather than generating hyperplanes sequentially from continuous data blocks—where each hyperplane differs from the previous by only one or two points—we generate a large set of candidate hyperplanes and then apply $sodt$ only to combinations in which the data points defining each splitting rule are sufficiently distinct—we want to only evaluate those that contain unique data points greater than a threshold $\alpha < D \times K$. This will help us to find sufficiently distinct hyperplanes.

E EXPERIMENTS

The experiments aim to provide a detailed analysis along four dimensions:

1. **Computational complexity and scalability**: We compare $sodt_{\text{vec}}$ and $sodt_{\text{rec}}$ under both sequential and parallelized settings, evaluating the scalability of the more efficient method.
2. **Empirical combinatorial complexity**: We analyze the combinatorial complexity of hyperplane and hypersurface decision tree models, demonstrating that the empirical search-space size—after safely filtering out fatal pairs—is substantially smaller than the theoretical upper bound provided in Part I.
3. **Analysis over synthetic datasets**: Using synthetic datasets generated from hyperplane decision trees (with data lying in convex polygon regions), we benchmark the performance of $hodtCoreset$. We systematically test the effects of ground truth tree size, data dimensionality, dataset size, label noise, and data noise separately. Our results show that the hyperplane decision tree model learned by our algorithm is not only more accurate in prediction but also more robust to noise.
4. **Generalization performance on real-world datasets**: We evaluate performance across 30 real-world datasets, comparing our hyperplane decision tree model learned by $sodtWSH$ against the state-of-the-art optimal decision tree algorithm Brita et al. (2025) and the well-known approximate algorithm CART. We demonstrate that, when model complexity is properly controlled, hyperplane decision trees consistently outperform axis-parallel models in out-of-sample tests.

Global optimality for the full HODT problem additionally requires exhaustive enumeration of the complete rule set generated by $genRules$, which becomes computationally intractable even for relatively small datasets. In contrast, $sodt_{\text{vec}}$ is exact for any fixed candidate rule set, because it exhaustively enumerates all proper size-K trees over that set.

The heuristic procedures $hodtCoreset$ and $sodtWSH$ reduce computation by selecting and filtering candidate rules before invoking the exact tree solver. Therefore, for each restricted candidate rule set passed to $sodt$ or $sodt_{\text{vec}}$, the returned tree is optimal within that restricted search space, but it is not necessarily globally optimal over the complete rule set generated by $genRules$. This is a practical trade-off imposed by the available computational resources. In principle, global optimality can be recovered by continuing candidate generation until the complete rule set has been explored, while caching previously evaluated rules and configurations to avoid redundant computation.

Also, due to the intractable combinatorics of HSDT, the experiments for points (3) and (4) are conducted exclusively for the $M = 1$ (hyperplane) case. To obtain high-quality solutions efficiently, we employ the two heuristic methods developed in Section D.

In these experiments, we adapt Brita et al. (2025)'s ConTree algorithm rather than Mazumder et al. (2022)'s Quant-BnB algorithm. We observed that ConTree often provides more accurate solutions while being considerably more efficient, making it more suitable for our comparisons. We note that

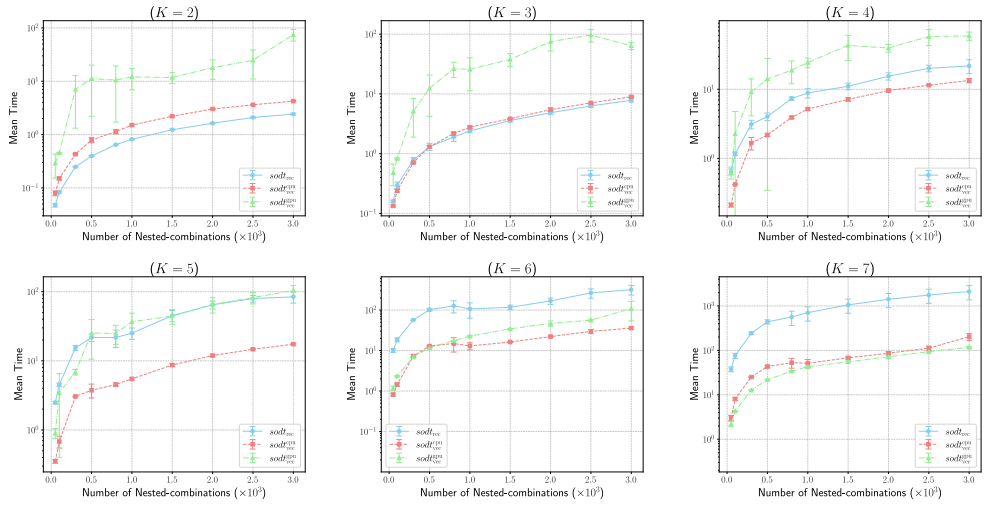


Figure 9: Running time comparison between $sodt_{\text{rec}}$ and $sodt_{\text{vec}}$ with varying K on sequential setting.

the current experiments do not explore the effects of hyperparameter tuning, such as minimum data per leaf or tree depth when optimizing size-constrained decision trees. The performance impact of these fine-grained controls represents an interesting avenue for future research.

All experiments were conducted on an Intel Core i9 CPU with 24 cores (2.4–6 GHz), 32 GB RAM, and a GeForce RTX 4060 Ti GPU.

E.1 Computational analysis

In this section, we analyze in detail the computational efficiency of $sodt$ for solving the size-constrained ODT problem. Since $sodt_{\text{vec}}$ and $sodt_{\text{kperms}}$ share similar advantages—both are fully vectorized but unable to exploit efficient dynamic programming (DP)—we focus on comparing $sodt_{\text{vec}}$ against the dynamic programming algorithm $sodt_{\text{rec}}$.

Rather than using a binary tree data structure, which relies on pointers to locate subtrees and suffers from poor cache performance, both $sodt_{\text{vec}}$ and $sodt_{\text{rec}}$ are implemented entirely with array (heap) data structures. This design ensures contiguous memory allocation, significantly reducing cache misses.

We compare their performance in two settings: 1) *Sequential setting*: Nested combinations are processed one-by-one using a standard for-loop. 2) *Parallelized setting*. A large batch of feasible nested combinations is processed simultaneously. Specifically, we store $ncrs : NCRs$ in a single large tensor instead of a list and pass it as input to $mapL\,(sodt, ncrs)$. For $mapL\,(sodt_{\text{vec}})$, we implement this as a single function (`batch_sodt`). This allows us to efficiently process the large batch of feasible nested combinations $ncrs$ on both CPU and GPU. We denote the results as $sodt_{\text{vec}}^{\text{cpu}}$ and $sodt_{\text{vec}}^{\text{gpu}}$, respectively.

In contrast, since $sodt_{\text{rec}}$ cannot be fully vectorized, we parallelize $mapL\,(sodt_{\text{vec}})$ using multi-core CPU execution, initialized in a multi-process setting.

E.1.1 Comparison between the vectorized and recursive implementation

Even though $sodt_{\text{rec}}$ has provably lower theoretical complexity, the hardware compatibility of $sodt_{\text{vec}}$ allows it to outperform in practice, particularly as K increases. As shown in Figure 9, in the sequential setting $sodt_{\text{rec}}$ is the most efficient method when $K \leq 3$. However, it becomes slower than $sodt_{\text{vec}}^{\text{cpu}}$ at $K = 4$, and eventually the slowest method once $K \geqq 5$. In contrast, $sodt_{\text{vec}}^{\text{gpu}}$ is initially slower than all other methods but overtakes $sodt_{\text{rec}}$ after $K = 5$, ultimately becoming the most efficient method at $K = 7$.

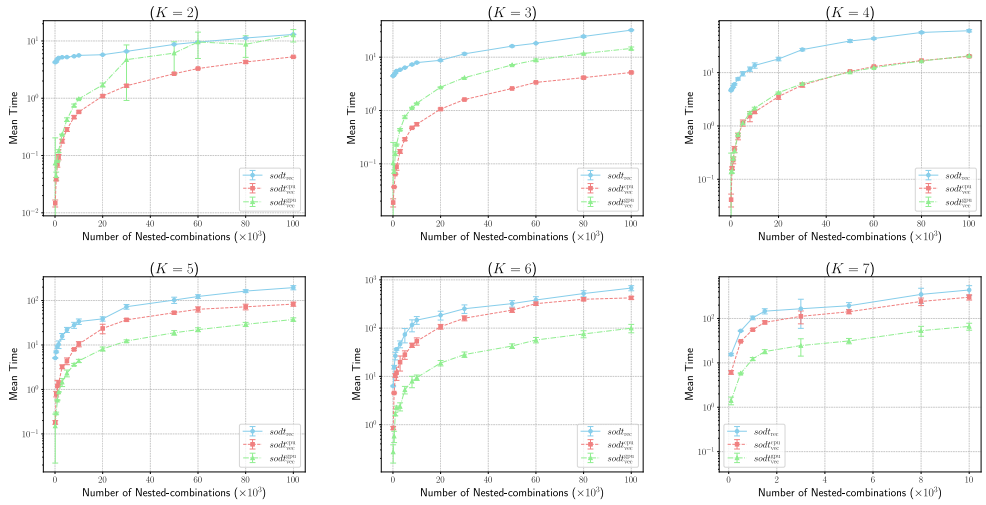


Figure 10: Running time comparison between $sodt_{\text{rec}}$ and $sodt_{\text{vec}}$ with varying K on parallel setting.

In the parallel setting (Figure 10), the benefits of vectorization are even more pronounced. Parallelizing $mapL\,(sodt_{\text{rec}})$ with multiprocessing incurs significant overhead from initializing multiple CPU cores, making $sodt_{\text{rec}}$ consistently the slowest method across all cases. Comparing $sodt_{\text{vec}}^{\text{cpu}}$ and $sodt_{\text{vec}}^{\text{gpu}}$, we observe that when $K < 4$, the CPU implementation is more efficient. For $K = 4$, both implementations achieve nearly identical performance, while for $K > 4$, $sodt_{\text{vec}}^{\text{gpu}}$ becomes superior. The GPU computation does not always outperform CPU computation because $mapL\,(sodt_{\text{vec}})$ in its current implementation consumes excessive memory to store intermediate results. The resulting memory transfer overhead can outweigh computational gains when k is small. A more sophisticated low-level implementation (e.g., in CUDA rather than Python) may reduce these overheads and further improve GPU performance.

E.1.2 Computational scalability of the vectorized method—the ability to explore one million nested combinations within a fixed time

Following the detailed comparison between $sodt_{\text{rec}}$ and $sodt_{\text{vec}}$ in both sequential and parallel settings, we now turn to the computational scalability of the winning method, $sodt_{\text{vec}}$. We focus on analyzing the pure computational scalability of the algorithm by benchmarking its performance without any acceleration techniques. Specifically, we measure the total wall-clock runtime of $sodt_{\text{vec}}$ for solving 1×10^6 instances of feasible nested combinations (containing no certified fatal pair) for different values of K. In our setup, cases with $K \leq 4$ are executed on the CPU, while cases with $K > 4$ are executed on the GPU.

As expected, when K is fixed, the bound in Theorem 9 is linear in N, and $sodt_{\text{vec}}$ exhibits approximately linear runtime growth. The batched vectorized implementation proves highly efficient for $K = 2, 3$; for example, it solves 1×10^6 feasible nested combinations in only a few tens of seconds. Note that a single size K nested combination can generate up to $K!$ possible proper decision trees in the worst-case. These results demonstrate the clear computational advantages of the $sodt_{\text{vec}}$ algorithm.

All current experiments are implemented in Python using the PyTorch library, and we anticipate that a lower-level implementation will further enhance performance.

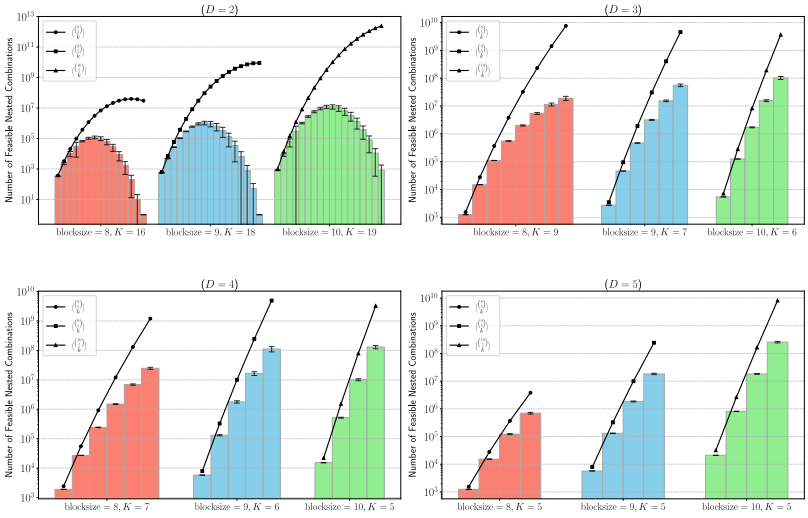


Figure 11: Combinatorial complexity of feasible nested combinations with varying K. Each bar corresponds to a fixed K starts from $K = 2$ (left-most bar) to the value of K labeled below each group of bars.

E.2 Combinatorial complexity of the hyperplane/hypersurface decision tree

E.2.1 Combinatorial complexity of nested combinations after filtering out fatal pairs

As noted, the combinatorial complexity of the hyperplane ($M = 1$) ODT problem is bounded by $\left(\binom{N}{D} \right) = O\left(N^{DK}\right)$. However, as established in Theorem 3, any nested combination containing a fatal pair cannot yield a complete decision tree. Consequently, all configurations containing a certified fatal pair can be filtered out without compromising the optimality of the algorithms.

To evaluate the true combinatorial complexity of the problem after excluding infeasible combinations, we conducted experiments on synthetic Gaussian datasets of size $N = 2000$. Due to memory limitations, we tested only subsets of the original dataset (denoted as "blocksize" in Figure 11. Specifically, a small sub-dataset was sampled from the $N = 2000$ dataset and then passed into $nestedCombsFA$, while hyperplane predictions were still carried out using the full $N = 2000$ dataset, but blocksize ranged from 8 to 10.

For each panel in Figure 11, we fixed the dimension D and varying N and K. We computed the combinatorial complexity over five datasets and reported the mean and standard deviation, illustrated as bar charts with error bars. The results show that the true combinatorial complexity (bars), after filtering infeasible nested combinations, is substantially smaller than the theoretical upper bound $O\left(N^{DK}\right)$ (black lines).For example, when blocksize $= 10$, $D = 2$, $K = 10$, the theoretical number of nested combinations is 3×10^9, whereas the number of feasible nested combinations is only 1×10^7, more than 300 times difference!

An interesting pattern emerges in the $D = 2$ panel: the complexity curve forms an inverted U-shape, indicating that the combinatorial complexity of decision trees initially increases with K but eventually decreases. This phenomenon aligns with the explanation given in He & Little (2025b), under mild probabilistic assumptions, the likelihood of constructing a feasible nested combination decreases exponentially with K. The inverted U-shape arises because, at first, the rapid growth in the number of nested combinations dominates, but beyond a certain threshold, the exponential decay in feasibility prevails. Nevertheless, in practice, this behavior is rarely observed, as it typically requires K to reach astronomically large values even for medium-sized datasets.

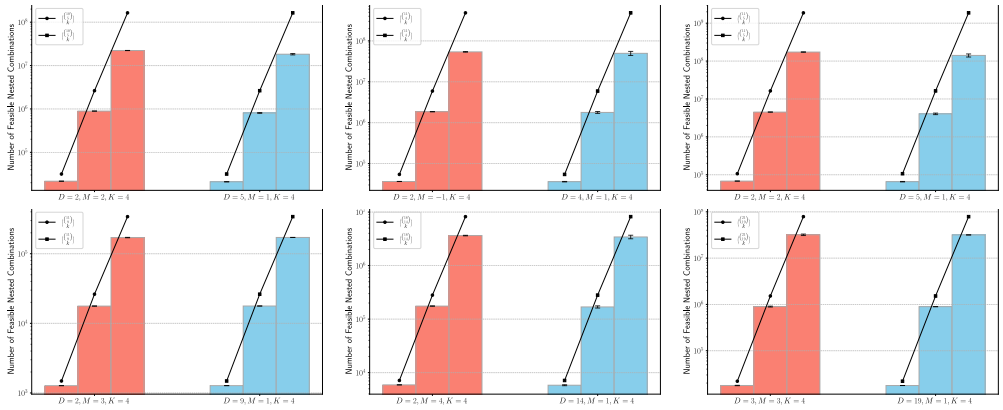


Figure 12: Combinatorial complexity of feasible nested combinations with varying K. Each bar corresponds to a fixed K starts from $K = 2$ (left-most bar) to the value of K labeled below each group of bars.

Comparing the combinatorial complexity of hypersurface and hyperplanes in the same dimension Having analyzed the combinatorial complexity of hyperplane splitting rules, we now extend the analysis to hypersurface rules by comparing their complexity with that of hyperplanes in the same dimension. As discussed, a hypersurface defined by a degree-M polynomial can be represented as a hyperplane embedded in a space of dimension $G = \binom{D+M}{D} - 1$ dimensional space.

For instance, when $D = 4$ and $M = 2$ we obtain $G = 14$. Our goal is to determine whether hypersurfaces in $\mathbb{R}^D$ (corresponding to hyperplanes in $\mathbb{R}^G$) exhibit different combinatorial complexity compared to hyperplanes. As illustrated in Figure 12, the degree-M hypersurface in $\mathbb{R}^D$ shows slightly higher combinatorial complexity than the corresponding hyperplane in $\mathbb{R}^G$, This increase may stem from the greater expressivity of hypersurfaces relative to hyperplanes.

E.3 Computational experiments with synthetic datasets

In this section, we evaluate the performance of the HODT model on a variety of synthetically generated datasets. The aim is to assess how effectively a more flexible model can recover the underlying ground truth, which is itself generated by a decision tree model.

Our experiments build on those of Murthy & Salzberg (1995) and Bertsimas & Dunn (2017). While their datasets were generated using axis-parallel decision trees, we generalize the setting by using hyperplane decision trees. Consequently, the underlying partitions in our datasets are polygonal regions. Specifically, we generate synthetic datasets from randomly constructed hyperplane decision trees, and then compare the performance of different algorithms in inducing trees that approximate the ground truth.

To construct the ground truth, we generate a decision tree of a specified size by choosing splits at random. The leaves of the tree are assigned unique labels so that no two leaves share the same label, ensuring the tree is the minimal representation of the ground truth. Training and test datasets are then generated by sampling each data point x uniformly at random and assigning its label according to the ground truth tree. In each experiment, **50** random trees are generated as ground truths, producing 50 corresponding training-test dataset pairs. The training set size varies across experiments, while the test set size is fixed a $2^d \times (D-1) \times 500$, where d is the depth of the tree and D s the data dimensionality.

Following Murthy & Salzberg (1995); Bertsimas & Dunn (2017), we evaluate tree quality using six metrics:

- **Training accuracy**: Accuracy on the training set.
- **Test accuracy**: Accuracy on the test set.

Tree size (K)	Method	Train Acc (%)	Test Acc (%)	Tree size	Depth		
					Maximum	Average	Expected
2	CART-depth-2	89.12 (0.04)	82.61 (0.04)	2.92 (0.27)	2.00 (0.00)	1.97 (0.09)	1.96 (0.15)
	CART-size-4	93.82 (3.01)	86.12 (3.73)	4.00 (0.00)	3.18 (0.38)	2.49 (0.16)	2.15 (0.20)
	ConTree-2	91.72 (3.83)	84.57 (4.03)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	100 (0.00)	93.50 (3.21)	2.00 (0.00)	2.00 (0.00)	1.67 (1.20)	1.65 (0.16)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (1.20)	1.66 (0.16)
3	CART-depth-3	90.72 (4.71)	81.30 (5.71)	5.90 (0.00)	3.00 (0.00)	2.83 (0.14)	2.77 (0.19)
	CART-size-6	93.88 (3.89)	83.92 (5.55)	6.00 (0.00)	4.02 (0.65)	3.10 (0.24)	2.71 (0.20)
	ConTree-3	95.98 (2.93)	85.25 (4.96)	6.94 (0.24)	3.00 (0.00)	2.99 (0.03)	2.99 (0.04)
	HDT	100 (0.00)	94.91 (2.36)	3.00 (0.00)	3.00 (0.00)	2.25 (0.00)	2.18 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	3.00 (0.00)	2.72 (0.45)	2.18 (0.11)	2.07 (0.24)
4	CART-depth-3	87.98 (5.26)	77.66 (5.38)	6.12 (0.86)	3.00 (0.00)	2.87 (0.14)	2.86 (0.15)
	CART-size-8	93.90 (3.93)	81.93 (4.93)	8.00 (0.00)	4.54 (0.57)	3.49 (0.21)	3.13 (0.21)
	ConTree-3	93.12 (4.02)	81.51 (4.78)	7.00 (0.00)	3.00 (0.00)	3.00 (0.00)	3.00 (0.00)
	HDT	97.46 (1.70)	89.89 (3.71)	4.00 (0.00)	3.52 (0.50)	2.63 (0.19)	2.60 (0.31)
	Ground Truth	100 (0.00)	100 (0.00)	4.00 (0.00)	3.52 (0.50)	2.63 (0.19)	2.63 (0.33)

Table 3: The effect of tree size of ground truth tree. No noise in data, training size = 100.

- **Tree size**: Number of branch nodes.
- **Maximum depth**: Maximum depth of the leaf nodes.
- **Average depth**: Mean depth of leaf nodes.
- **Expected depth**: Average depth of leaf nodes weighted by the proportion of the test set samples assigned to each leaf.

We also follow Bertsimas & Dunn (2017) in examining the effect of five sources of variation or noise: 1) Ground-truth tree size, 2)Training data size, 3)Label noise, 4) Feature noise, 5) Data dimensionality.

Since Bertsimas & Dunn (2017)'s implementation is not publicly available, and replicating their construction is difficult, so we benchmark against three widely used axis-parallel decision tree algorithms: CART-depth (depth-constrained CART algorithm), CART-size (size-constrained CART algorithm), and the state-of-the-art optimal axis-parallel ODT algorithm—ConTree (Brita et al., 2025) (which only allows tree depth to be specified). For all experiments in this section, we employ the $hodtCoreset$ algorithm for generating accurate hypersurface decision tree.

Effect of tree size In our first experiments, we evaluated the effectiveness of each method as problem complexity increased. We fixed the training set size at $N = 100$ and $D = 2$, while varying the ground truth size K from 2 to 4. Table 3 presents the results. Since our goal was to solve the size-constrained ODT problem, we set the depth of the depth-constrained algorithms as $d = \lceil \log_2 (K \times D) \rceil$ for the CART-depth and ConTree algorithms. CART-size constrained the tree size to $K \times D$. Note that a depth d tree has at most $2^d - 1$ tree size.

The results show that our algorithm has a very high probability of finding the optimal solution (100% for $K = 2$ and $K = 3$) and HDT significantly outperforms the other methods. Although all methods show a decrease in both training and test accuracy as K increase to 4, this decline is expected due to the increasing problem complexity. Even so, HDT and ConTree are more robust to changes in tree size. When K increases to 4, CART-size and CART-depth exhibit a larger reduction in out-of-sample accuracy (4.19% and 4.95%, respectively) compared to ConTree and HDT, which show smaller decreases (3.06% and 3.61%). Interestingly, although the optimal axis-parallel decision tree obtained by the ConTree algorithm is more accurate than CART-depth under the same depth constraint, ConTree often fully exploits the given depth, resulting in nearly maximal tree sizes in most cases. Consequently, ConTree typically produces larger trees than CART-depth. However, CART-size, by allowing just one additional leaf, achieves better solutions than ConTree by providing greater flexibility in tree depth.

In summary, the experiments demonstrate that HDT is not only more accurate than CART and ConTree across varying tree sizes and depths, but also closely matches the ground truth across all

measures. In contrast, axis-parallel trees learned using CART and ConTree are significantly larger than HDT trees and produce substantially worse solutions.

Training set size	Method	Train Acc (%)	Test Acc (%)	Tree size	Depth		
					Maximum	Average	Expected
100	CART-depth-2	89.76 (5.11)	75.46 (15.70)	2.88 (0.32)	2.00 (0.00)	1.96 (0.11)	1.96 (0.12)
	CART-size-4	97.62 (2.51)	80.26 (16.80)	4.00 (0.00)	3.92 (0.69)	3.01 (0.30)	2.52 (0.33)
	ConTree-2	93.52 (3.53)	78.62 (16.14)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	100 (0.00)	84.38 (19.27)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.66 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	0.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)
200	CART-depth-2	89.00 (4.49)	84.62 (5.23)	2.92 (0.27)	2.00 (0.00)	1.97 (0.09)	1.98 (0.07)
	CART-size-4	94.49 (3.02)	89.57 (4.04)	4.00 (0.00)	3.26 (0.44)	2.50 (0.17)	2.17 (0.21)
	ConTree-2	92.87 (3.38)	88.51 (3.99)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.94 (0.16)	97.97 (1.01)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.71 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)
400	CART-depth-2	87.45 (4.69)	84.62 (4.90)	2.98 (0.14)	2.00 (0.00)	1.99 (0.05)	1.99 (0.05)
	CART-size-4	92.58 (3.48)	89.35 (4.08)	4.00 (0.00)	3.24 (0.43)	2.53 (0.17)	2.15 (0.24)
	ConTree-2	91.44 (3.37)	88.78 (4.00)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.85 (0.21)	98.98 (0.58)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.71 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)
800	CART-depth-2	86.91 (4.63)	85.14 (4.96)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	92.21 (3.43)	90.12 (3.82)	4.00 (0.00)	3.3 (0.00)	2.53 (0.18)	2.13 (0.22)
	ConTree-2	91.00 (3.59)	89.18 (3.94)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.87 (0.14)	99.40 (0.36)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.69 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)
1600	CART-depth-2	85.66 (5.05)	84.97 (5.15)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	91.48 (3.69)	90.53 (3.71)	4.00 (0.00)	3.18 (0.38)	2.51 (0.16)	2.16 (0.26)
	ConTree-2	90.13 (3.69)	89.38 (3.84)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.87 (0.12)	99.65 (0.22)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.24)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)

Table 4: The effect of training data size. Ground truth-tree size $K = 2$.

Effect of data size The second set of experiments demonstrates the effect of training data size relative to problem complexity. We increased the size of the training set while keeping the ground truth size fixed at $K = 2$ and the dimension at $D = 2$. ConTree and CART-depth were restricted to depth 2, while CART-size was restricted to $K \times D = 4$.

No noise was added to the data. Table 4 shows that out-of-sample accuracy increased for all methods as the training set grew. The improvement was most pronounced for HDT, which achieved a 15.27% increase in out-of-sample accuracy, whereas ConTree, CART-size, and CART-depth improved by 10.76%, 10.27%, and 9.51%, respectively. Notably, CART-depth performed significantly worse than the other methods.

These results demonstrate that even in data-poor environments, optimizing an appropriate model (HDT in this case) substantially improves out-of-sample performance. Even when axis-parallel trees do not match the ground truth, optimizing the solution to optimality still produces significant differences. The optimal algorithm (ConTree) achieves much higher test accuracy than the approximate method (CART-depth) on both training and test datasets. This observation is consistent with Bertsimas & Dunn (2017), and our experiments provide additional evidence for scenarios in which the ground truth does not align with the chosen model. These results offer clear support against the notion that optimal methods necessarily overfit the training data in data-scarce settings.

Effect of data dimension The third set of experiments examines the effect of problem dimensionality while keeping the tree size ($K = 2$) and training set size ($N = 100$) constant. ConTree and CART-depth were fixed at depth 2, and CART-size was fixed at size 4. Table 5 shows the effect of increasing the number of features for a fixed training size and tree size.

Although HDT might be expected to suffer most from the increased combinatorial complexity associated with higher dimensionality, it remains the most robust method on both training and test datasets, exhibiting the smallest decrease in accuracy. Increasing the number of features significantly affects all methods: CART-depth experiences decreases of approximately 18.61% in training

Data dimension	Method	Train Acc (%)	Test Acc (%)	Tree size	Depth		
					Maximum	Average	Expected
2	CART-depth-2	89.00 (4.49)	84.62 (5.23)	2.92 (0.27)	2.00 (0.00)	1.97 (0.09)	1.98 (0.07)
	CART-size-4	94.49 (3.02)	89.57 (4.04)	4.00 (0.00)	3.26 (0.44)	2.53 (0.17)	2.17 (0.21)
	ConTree-2	92.87 (3.38)	88.51 (3.99)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.0 (0.00)
	HDT	99.94 (0.16)	97.97 (1.01)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.71 (0.25)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.70 (0.25)
4	CART-depth-2	79.94 (4.53)	73.97 (4.74)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	84.17 (4.41)	76.72 (4.48)	4.00 (0.00)	3.10 (0.3)	2.48 (0.13)	2.24 (0.18)
	ConTree-2	82.80 (4.16)	76.16 (4.82)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	95.13 (1.38)	90.97 (2.55)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.13)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.12)
8	CART-depth-2	75.05 (5.72)	67.54 (7.08)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	79.25 (5.66)	69.66 (6.75)	4.00 (0.00)	3.12 (0.325)	2.48 (0.14)	2.21 (0.22)
	ConTree-2	77.78 (5.36)	69.40 (6.37)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	89.08 (4.26)	82.95 (5.70)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.48 (0.12)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.49 (0.13)
12	CART-depth-2	70.39 (3.82)	62.78 (5.20)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	74.50 (4.00)	64.03 (4.99)	4.00 (0.00)	3.14 (0.35)	2.46 (0.14)	2.26 (0.13)
	ConTree-2	73.35 (3.49)	63.53 (4.78)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	85.06 (5.19)	76.71 (6.47)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.50 (0.11)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.52 (0.10)

Table 5: Effects of dimensionality. Training size = 100. No noise. Ground truth tress are size 2.

accuracy and 21.84% in test accuracy; CART-size decreases by 19.99% (train) and 25.54% (test); ConTree decreases by 19.52% (train) and 24.98% (test); whereas HDT decreases only by 14.88% (train) to 21.26% (test). This robustness may be attributed to the effectiveness of $hodtCoreset$, which efficiently explores configurations without being heavily affected by combinatorial complexity.

Interestingly, our results for axis-parallel ODT algorithms on datasets with hyperplane decision tree ground truth differ from those reported by Bertsimas & Dunn (2017). However, it is important to note that our experimental setup differs from theirs. Bertsimas & Dunn (2017) observed that the performance gap between axis-parallel ODT algorithms and approximate CART increases in higher dimensions, with little difference in lower dimensions. In contrast, our experiments show that at lower dimensions, there is a significant difference between CART-depth and ConTree (3.89%), which decreases to 0.75% when $D = 12$.

Effect of noise on labels In the fourth set of experiments, we introduced noise in the labels. As the noise level increased, it was noteworthy that HDT was able to find solutions that outperformed the ground truth on the training dataset. As before, we fixed $D = 2$ and ($K = 2$). Following the experimental setup of Bertsimas & Dunn (2017) we added noise by increasing the label of a random $k\%$ of the points by 1, where Table 6 presents the results.

As the noise level increased, the accuracy of all methods tended to decrease, with similar effects on out-of-sample performance up to a noise level of 20%. Beyond this point (from 20% to 25% noise), differences became more pronounced: CART-depth, CART-size, and ConTree decreased by 1.47%, 1.73%, and 1.55%, respectively, whereas HDT decreased by only 0.91%. From a broader perspective, when increasing noise from 0% to 25%, HDT's out-of-sample accuracy decreased by 2.39%, compared with 2.93%, 3.43%, and 2.66% for ConTree, CART-size, and CART-depth, respectively. Notably, starting at a 5% noise level, HDT was able to find solutions that exceeded the ground truth on the training dataset while maintaining strong performance on out-of-sample tests. These results further refute the notion that optimal methods are less robust to noise, highlighting the superiority of HDT.

Effect of noise on data Finally, in the last set of experiments, we examined the effect of noise in the features of the data. We fixed $N = 100$, $D = 2$ and $K = 2$. Again, HDT was able to find solutions that outperformed the ground truth on the training dataset while achieving the best out-of-sample performance. Combined with previous experiments, these results strongly refute the idea that optimal algorithms necessarily overfit the data. Even when training accuracy exceeds that of the ground truth, proper control of model complexity prevents overfitting.

Noise level	Method	Train Acc (%)	Test Acc (%)	Tree size	Depth		
					Maximum	Average	Expected
0	CART-depth-2	90.42 (4.61)	87.25 (5.20)	2.92 (0.27)	2.00 (0.00)	1.97 (0.09)	1.99 (0.03)
	CART-size-4	95.19 (3.11)	91.21 (3.94)	4.0 (0.00)	3.18 (0.38)	2.49 (0.16)	2.11 (0.23)
	ConTree-2	93.49 (3.60)	89.99 (3.71)	2.98 (0.04)	2.00 (0.00)	1.99 (0.05)	1.98 (0.11)
	HDT	99.62 (0.68)	98.06 (1.21)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.28)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
5	CART-depth-2	86.29 (4.48)	87.15 (4.89)	2.94 (0.24)	2.00 (0.00)	1.98 (0.08)	1.99 (0.02)
	CART-size-4	90.64 (3.07)	91.01 (3.67)	4.0 (0.00)	3.20 (0.40)	2.50 (0.16)	2.10 (0.24)
	ConTree-2	89.07 (3.45)	89.87 (3.47)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	95.00 (0.20)	97.20 (1.40)	2.00 (0.00)	2.00 (0.00)	1.68 (0.00)	1.62 (0.29)
	Ground Truth	95.00 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
10	CART-depth-2	82.25 (3.78)	86.74 (5.36)	2.98 (0.14)	2.00 (0.00)	1.99 (0.05)	2.00 (0.01)
	CART-size-4	86.25 (2.67)	90.45 (4.07)	4.00 (0.00)	3.12 (0.32)	3.12 (0.32)	2.16 (0.26)
	ConTree-2	84.69 (3.09)	89.48 (3.96)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	90.45 (0.49)	97.59 (1.21)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.64 (0.30)
	Ground Truth	90.00 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
15	CART-depth-2	77.90 (3.99)	85.99 (5.71)	2.98 (0.14)	2.00 (0.00)	1.99 (0.05)	2.00 (0.04)
	CART-size-4	81.80 (3.10)	90.19 (3.67)	4.00 (0.00)	3.22 (0.41)	2.51 (0.17)	2.19 (0.31)
	ConTree-2	80.53 (3.04)	89.10 (3.79)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	85.69 (0.54)	97.10 (1.84)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.63 (0.29)
	Ground Truth	84.99 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
20	CART-depth-2	73.88 (3.16)	86.06 (4.95)	2.94 (0.24)	2.00 (0.00)	1.98 (0.08)	2.00 (0.01)
	CART-size-4	77.36 (2.91)	89.51 (4.59)	4.00 (0.00)	3.24 (0.43)	2.54 (0.17)	2.13 (0.34)
	ConTree-2	76.28 (2.81)	88.61 (4.43)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	80.98 (0.69)	96.58 (2.06)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.27)
	Ground Truth	79.90 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)
25	CART-depth-2	70.35 (3.22)	84.59 (6.66)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	73.19 (2.54)	87.78 (5.41)	4.00 (0.00)	3.26 (0.44)	2.52 (0.17)	2.24 (0.32)
	ConTree-2	72.26 (2.62)	87.06 (5.16)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	76.37 (0.83)	95.67 (2.93)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.78)
	Ground Truth	75.00 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.68 (0.28)

Table 6: The effect of noise on labels. Training data size = 100. Ground truth trees are size 2.

In summary, across all synthetic data experiments, HDT-generated trees most closely matched the quality metrics of the ground truth trees. HDT not only produced more accurate results but also generated smaller trees, demonstrating a clear advantage in scenarios where tree size must be strictly controlled. Interestingly, by allowing slightly more flexibility in size or depth, CART can sometimes achieve slightly better solutions than optimal axis-parallel decision tree algorithms. However, in contexts requiring strict control of tree size, HDT consistently provides superior performance compared to all other methods.

Moreover, our experiments align with the observations of Bertsimas & Dunn (2017), countering the widely held misconception that optimal methods are more prone to overfitting the training set at the expense of out-of-sample accuracy. We demonstrate that optimal solutions remain robust in data-scarce and noisy settings, even when training accuracy exceeds that of the ground truth.

E.4 Computational experiments on real-world datasets

We now present a direct comparison between HDT, ConTree, CART-depth, and CART-size on real-world datasets. Our goal is to evaluate the effectiveness of the more flexible hyperplane decision tree model relative to the standard axis-parallel decision tree models. Table 8 reports the mean out-of-sample accuracy for 30 classification datasets from the UCI Machine Learning Repository.

As expected, when model complexity is controlled, HDT outperforms the axis-parallel methods on most datasets. Specifically, HDT achieves higher training accuracy on 25 datasets and better out-of-sample accuracy on 21 datasets over ConTree. By contrast, ConTree achieves the best training accuracy on only 4 datasets and the best out-of-sample accuracy on 6 datasets. The CART-depth algorithm achieves the highest out-of-sample accuracy on just 3 datasets, performing clearly worse than the optimal ConTree under the same constraints. Even with greater flexibility (allowing tree depths beyond 2), CART-depth achieves the highest accuracy on only 5 datasets, still falling behind HDT.

Noise level	Method	Train Acc (%)	Test Acc (%)	Tree size	Depth		
					Maximum	Average	Expected
0	CART-depth-2	90.29 (4.74)	87.23 (5.14)	2.96 (0.20)	2.00 (0.00)	1.99 (0.065)	1.99 (0.09)
	CART-size-4	94.20 (3.70)	90.32 (4.24)	4.00 (0.00)	3.2 (0.399)	2.51 (0.16)	2.09 (0.33)
	ConTree-2	92.73 (3.90)	89.55 (4.10)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.88 (0.26)	98.08 (1.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.63 (0.29)
	Ground Truth	100 (0.00)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.63 (0.28)
5	CART-depth-2	90.16 (4.63)	87.16 (5.06)	2.98 (0.14)	2.00 (0.00)	1.99 (0.05)	2.00 (0.01)
	CART-size-4	93.98 (3.78)	90.32 (4.20)	4.00 (0.00)	3.24 (0.43)	2.53 (0.17)	2.05 (0.32)
	ConTree-2	92.62 (3.79)	89.50 (3.94)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.62 (0.40)	98.10 (1.01)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.64 (0.28)
	Ground Truth	99.59 (0.44)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.65 (0.28)
10	CART-depth-2	90.30 (4.55)	87.36 (5.09)	2.96 (0.20)	2.00 (0.00)	1.99 (0.065)	1.99 (0.09)
	CART-size-4	93.98 (3.57)	90.30 (4.55)	4.00 (0.00)	3.28 (0.45)	2.54 (0.17)	2.03 (0.34)
	ConTree-2	92.50 (3.65)	89.38 (4.09)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.07 (0.62)	97.50 (0.90)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.58 (0.29)
	Ground Truth	98.81 (0.71)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.65 (0.28)
15	CART-depth-2	89.86 (4.91)	87.06 (4.97)	2.96 (0.20)	2.00 (0.00)	1.99 (0.07)	2.00 (0.02)
	CART-size-4	93.78 (3.92)	90.27 (4.42)	4.00 (0.00)	3.32 (0.47)	2.55 (0.18)	2.03 (0.33)
	ConTree-2	92.52 (3.83)	89.31 (4.38)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	99.05 (0.66)	97.90 (1.10)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.63 (0.28)
	Ground Truth	98.72 (0.80)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.65 (0.28)
20	CART-depth-2	89.75 (4.98)	86.61 (5.57)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	CART-size-4	93.75 (3.70)	90.21 (4.27)	4.00 (0.00)	3.18 (0.38)	2.49 (0.16)	2.08 (0.27)
	ConTree-2	92.39 (3.72)	89.25 (4.05)	3.00 (0.00)	2.00 (0.00)	2.00 (0.00)	2.00 (0.00)
	HDT	98.65 (0.87)	97.40 (11.50)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.64 (0.28)
	Ground Truth	98.11 (1.04)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.65 (0.28)
25	CART-depth-2	89.76 (4.47)	87.13 (4.93)	2.98 (0.14)	2.00 (0.00)	1.99 (0.046)	1.99 (0.09)
	CART-size-4	93.58 (4.13)	90.28 (4.50)	4.00 (0.00)	3.24 (0.43)	2.53 (0.17)	2.05 (0.35)
	ConTree-2	92.29 (3.65)	89.55 (4.17)	2.98 (0.00)	2.00 (0.00)	1.99 (0.05)	1.99 (0.05)
	HDT	98.35 (1.00)	97.62 (1.20)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.67 (0.27)
	Ground Truth	97.67 (0.91)	100 (0.00)	2.00 (0.00)	2.00 (0.00)	1.67 (0.00)	1.65 (0.28)

Table 7: The effect of noise on training data. Training data size = 100. Ground truth trees are size 2.

Dataset	N	D	C	CART-depth $d = 2$	ConTree $d = 2$	CART-size $K = 2$	CART-size $K = 3$	HDT $K = 2$	HDT $K = 3$
Parkinsons	195	22	2	87.95/84.62 (1.48/5.85)	93.20/82.56 (1.04/5.23)	87.82/85.13 (1.67/6.36)	90.77/86.67 (1.04/5.71)	93.72/**89.74** (1.66/3.14)	**94.49/89.74** (2.15/3.14)
Diabetic	1146	19	2	65.55/64.00 (0.92/3.74)	67.36/63.48 (0.79/3.95)	65.55/64.00 (0.92/3.74)	67.53/62.96 (1.05/2.66)	79.93/**76.52** (1.17/1.71)	**80.09**/76.35 (1.06/2.38)
BalScl	625	4	3	71.72/65.12 (0.94/2.00)	72.88/66.88 (0.35/1.40)	69.80/63.20 (0.44/1.75)	72.68/64.64 (1.12/2.74)	93.44/94.40 (0.33/1.26)	**94.28/94.56** (0.30/1.54)
StatlogVS	845	18	4	53.34/52.66 (0.32/2.97)	62.42/61.89 (0.91/4.07)	49.94/48.40 (0.64/3.73)	53.34/52.66 (0.32/2.97)	65.12/63.31 (0.78/4.73)	**68.75/64.38** (0.74/2.58)
ImgSeg	210	19	7	50.00/41.91 (6.42/9.71)	58.69/49.04 (0.80/3.23)	44.64/33.81 (1.30/5.51)	57.86/**52.38** (1.09/4.26)	66.00/41.34 (2.58/6.24)	**66.47**/49.52 (1.91/5.20)
iris	147	4	3	97.09/90.67 (0.68/2.49)	97.09/90.66 (0.68/2.49)	97.09/90.67 (0.68/2.49)	98.12/91.33 (1.26/3.40)	**99.80/97.33** (3.49/7.14)	98.97/96.77 (4.18/6.31)
MnkPrb	432	6	2	74.44/77.24 (1.53/6.06)	78.38/75.40 (1.49/5.89)	74.44/77.24 (1.53/6.06)	81.39/82.76 (3.28/6.86)	82.24/79.75 (1.56/3.96)	**82.26/86.20** (2.79/2.44)
UKM	403	5	5	81.18/75.80 (1.52/4.18)	**82.17/76.05** (0.54/1.84)	78.14/75.31 (0.72/4.18)	82.05/75.80 (0.50/2.59)	71.24/64.69 (9.41/16.69)	72.42/68.15 (10.74/16.84)
TchAsst	106	5	4	54.76/39.09 (1.68/5.46)	57.14/41.81 (1.30/5.30)	53.33/40.00 (0.48/4.45)	56.43/42.73 (2.33/4.64)	71.19/57.27 (10.0/7.61)	**75.71/60.90** (1.60/9.96)
RiceCammeo	3810	7	2	92.91/92.52 (0.15/0.67)	93.30/92.91 (0.19/0.80)	92.91/92.52 (0.15/0.67)	92.91/92.52 (0.15/0.67)	94.00/93.28 (0.29/1.28)	**94.18/93.31** (0.29/1.39)
Yeast	1453	8	10	48.80/45.77 (0.35/0.83)	49.81/**49.28** (0.18/1.55)	46.56/44.19 (0.43/0.77)	48.80/45.77 (0.35/0.83)	48.62/45.66 (0.66/1.38)	**49.60**/45.57 (0.70/3.09)
WineQuality	5318	11	7	52.96/52.88 (0.39/0.45)	54.07/53.85 (0.27/1.07)	52.45/52.59 (0.93/0.58)	52.96/52.88 (0.39/0.45)	54.70/**55.23** (0.31/0.98)	**55.09/55.23** (0.29/1.16)
SteelOthers	1941	27	2	70.59/71.11 (0.21/1.32)	73.75/71.31 (0.31/0.85)	70.50/71.05 (0.36/1.26)	72.35/72.55 (0.89/1.21)	77.43/**76.52** (0.05/0.81)	**78.09**/75.83 (1.15/1.98)
DrgCnsAlc	1885	12	7	69.07/**69.87** (0.48/1.92)	69.54/69.66 (0.39/1.90)	69.07/**69.87** (0.48/1.92)	69.07/**69.87** (0.48/1.92)	70.89/69.07 (0.35/1.33)	**71.70**/69.23 (0.36/1.48)
DrgCnsImp	1885	12	7	40.41/39.79 (0.57/2.03)	41.95/38.04 (0.39/0.87)	40.36/39.89 (0.55/2.19)	40.41/39.79 (0.57/2.03)	45.04/42.55 (0.88/3.07)	**45.73/42.60** (2.39/1.80)
DrgCnsSS	1885	12	7	51.54/51.88 (0.36/2.75)	52.73/52.20 (0.35/1.70)	51.53/**52.79** (0.35/1.38)	51.54/51.88 (0.36/2.75)	54.13/52.31 (0.58/1.70)	**54.93**/52.10 (0.51/1.28)
EstObLvl	2087	16	7	55.36/**55.98** (0.18/0.66)	**55.40**/55.31 (0.14/1.13)	43.14/42.39 (0.30/1.33)	55.36/**55.98** (0.18/0.66)	46.05/44.21 (1.09/2.87)	51.99/50.28 (1.82/2.64)
AiDS	2139	23	2	85.44/85.70 (0.39/01.43)	87.54/86.73 (0.11/0.58)	85.44/85.70 (0.39/1.43)	**89.18/88.51** (0.20/0.68)	86.90/85.79 (1.24/1.93)	87.13/85.65 (1.31/1.94)
AucVer	2043	7	2	90.42/**89.58** (0.55/1.57)	**91.00/89.98** (0.40/1.59)	90.27/89.49 (0.40/1.61)	90.42/**89.58** (0.55/1.57)	89.65/88.46 (0.35/1.99)	89.87/88.66 (0.34/1.83)
Ai4iMF	10000	6	2	97.25/97.07 (0.11/0.36)	97.39/97.31 (0.07/0.26)	97.08/97.00 (00.27/00.34)	97.16/97.03 (0.22/0.36)	97.79/97.95 (0.11/0.18)	**97.90/98.00** (0.12/0.15)
VoicePath	704	2	2	96.87/95.60 (0.51/2.12)	97.16/96.31 (0.56/2.43)	96.87/95.60 (0.51/2.12)	97.16/95.04 (0.48/1.85)	97.69/**97.44** (0.28/1.08)	**97.90**/97.31 (0.32/1.17)
WaveForm	5000	21	3	70.69/68.50 (0.25/1.33)	**71.28/68.54** (0.34/0.81)	66.31/65.02 (0.74/0.87)	70.69/68.50 (0.25/1.33)	67.05/66.62 (2.44/1.55)	65.12/65.80 (1.82/1.65)

Table 8: (Part II: K in 2, 3) Five-fold cross-validation results on the UCI dataset. We compare the performance of our HDT algorithm, with K (number of splitting rules) ranging from 2 to 3, trained using $sodtWSH$—against approximate methods:size- and depth-constrained CART algorithms (CART-size and CART-depth), as well as the state-of-the-art optimal axis-parallel decision tree algorithm, ConTree. The depth of the CART-depth and ConTree algorithms are fixed at 2. Results are reported as mean accuracy on the training and test sets in the format *Training Accuracy / Test Accuracy(Standard Deviation: Train / Test)*. The best-performing algorithm in each row is shown in **bold**.

Dataset	N	D	C	CART-depth $d=3$	ConTree $d=3$	CART-size $K=4$	CART-size $K=5$	CART-size $K=6$	HDT $K=4$	HDT $K=5$	HDT $K=6$
haberman	283	3	2	77.35/74.39 (1.38/3.25)	80.97/72.98 (1.22/03.94)	76.20/73.33 (1.99/2.58)	77.17/72.28 (2.21/1.72)	77.97/72.28 (1.71/2.58)	81.68/78.94 (0.86/3.51)	82.48/79.29 (0.97/4.30)	**82.83/79.65** (1.15/4.37)
BldTrns	502	4	2	77.41/74.06 (0.64/03.62)	80.45/74.06 (0.843/2.76)	78.40/77.23 (0.56/2.80)	79.05/75.45 (1.32/4.13)	79.20/73.86 (1.21/3.41)	81.68/78.94 (0.97/3.51)	82.48/79.30 (1.20/2.88)	**82.83/79.65** (1.15/4.23)
sepsis	975	3	2	94.23/93.74 (0.35/1.19)	94.56/93.44 (0.34/1.10)	94.28/93.54 (0.33/1.11)	94.39/93.54 (0.34/0.95)	94.44/93.54 (0.29/0.95)	95.77/93.85 (0.43/1.95)	96.10/93.85 (0.43/1.95)	**96.44/94.05** (0.46/1.72)
algerian	243	14	2	99.59/98.37 (0.21/1.53)	**100**/95.51 (0.00/3.00)	99.79/98.37 (0.25/1.53)	99.90/98.37 (0.21/1.53)	**100**/98.37 (0.00/1.53)	**100/99.18** (0.00/1.12)	**100**/97.96 (0.00/1.44)	**100**/96.74 (0.00/1.83)
Cryotherapy	89	6	2	94.93/90.00 (1.91/06.48)	99.44/78.89 (0.69/12.86)	94.93/90.00 (1.91/6.48)	96.62/86.67 (1.91/9.03)	97.75/86.67 (1.69/9.03)	99.44/**94.44** (0.77/3.93)	**99.72**/93.33 (0.77/3.93)	**99.72**/92.22 (0.49/1.95)
Caesarian	72	5	2	77.90/58.67 (1.79/7.78)	82.11/58.67 (1.72/4.99)	77.54/66.67 (2.33/4.22)	78.60/56.00 (3.58/12.36)	80.35/57.33 (2.58/6.80)	92.63/85.33 (1.47/2.98)	**93.68/86.68** (20.96/0.00)	**93.68**/85.33 (20.96/2.98)
ecoli	336	7	8	85.52/81.47 (0.37/1.77)	**87.84**/82.06 (0.18/2.85)	85.07/81.76 (0.53/1.50)	85.82/82.65 (0.88/2.53)	85.97/**82.94** (0.65/2.20)	84.25/79.12 (1.70/4.08)	84.40/79.41 (21.25/4.29	85.90/79.12 (21.25/4.46)
GlsId	213	9	6	72.35/66.05 (1.86/6.51)	**80.12/70.70** (0.941/3.78)	71.29/66.98 (1.72/6.14)	74.24/63.72 (1.46/4.79)	75.88/64.65 (2.10/7.98)	76.35/61.86 (1.97/4.53)	77.77/61.86 (21.92/4.53)	77.29/58.61 (21.64/4.47)

Table 9: (Part I: K in 4, 5, 6) Five-fold cross-validation results on the UCI dataset. We compare the performance of our HDT algorithm, with K (number of splitting rules) ranging from 4 to 6, trained using $sodtWSH$—against approximate methods:size- and depth-constrained CART algorithms (CART-size and CART-depth), as well as the state-of-the-art optimal axis-parallel decision tree algorithm, ConTree. The depth of the CART-depth and ConTree algorithms are fixed at 3. Results are reported as mean accuracy on the training and test sets in the format *Training Error / Test Error (Standard Deviation: Train / Test)*. The best-performing algorithm in each row is shown in **bold**.

In summary, the more flexible hyperplane decision tree model shows strong potential for interpretable learning tasks under controlled model complexity, often delivering substantially better performance than axis-parallel models. For instance, on the BalScl dataset, HDT improves training accuracy by over 20% and test accuracy by nearly 30% compared to the optimal axis-parallel tree algorithm, ConTree. These results demonstrate the benefit of moving beyond axis-parallel splitting rules: even small hyperplane decision trees can represent decision boundaries that require substantially larger axis-parallel trees. Quantitative evaluation of higher-degree hypersurface trees remains future work; in the present paper, their empirical behavior is demonstrated only on small synthetic examples because of the rapidly growing candidate space.

E.4.1 Experiments for $K > 3$ decision trees

The experimental results for $K > 3$ decision trees are given in Table 9 and 10.

Dataset	N	D	C	CART-depth $d = 3$	ConTree $d = 3$	CART-size $K = 4$	CART-size $K = 5$	CART-size $K = 6$	HDT $K = 4$	HDT $K = 5$	HDT $K = 6$
iris	147	4	3	98.12/90.67 (1.26/2.49)	99.82/91.33 (0.34/4.00)	98.80/88.00 (1.03/4.52)	99.15/88.00 (0.76/4.52)	99.32/88.00 (0.64/4.52)	99.15/96.77 (2.94/12.61)	**100/97.77** (0.00/12.61)	**100**/96.77 (0.00/12.61)
MnkPrb	432	6	2	82.44/78.62 (3.61/3.89)	**89.51/86.44** (1.43/5.65)	82.43/78.62 (3.61/3.89)	83.25/78.16 (2.41/2.72)	81.54/77.01 (2.60/3.17)	82.46/81.38 (1.30/5.6)	82.73/81.61 (1.61/5.75)	82.96/82.30 (1.50/6.38)
UKM	403	5	5	88.76/86.42 (1.38/3.75)	**91.55/87.90** (0.60/3.26)	85.34/80.00 (0.46/3.86)	87.39/83.70 (0.64/3.44)	89.57/87.65 (0.32/1.35)	73.10/68.89 (10.74/17.00)	73.60/75.30 (10.17/9.88)	73.79/75.56 (10.3/9.54)
TchAsst	106	5	4	62.38/41.82 (2.21/4.45)	67.62/46.36 (2.65/10.52)	58.81/40.91 (1.43/4.98)	62.14/43.64 (2.31/3.64)	63.81/42.73 (2.88/4.64)	78.33/**62.73** (1.13/10.85)	79.76/**62.73** (1.19/10.85)	**80.48**/59.09 (1.81/8.50)
RiceCammeo	3810	7	2	93.06/92.49 (0.28/0.79)	93.79/92.36 (0.16/0.89)	92.91/92.52 (0.15/0.67)	93.03/92.49 (0.28/0.72)	93.09/92.49 (0.31/0.76)	94.27/93.49 (0.29/1.36)	94.35/**93.57** (0.29/1.31)	**94.38/93.57** (0.31/1.20)
Yeast	1453	8	10	57.86/54.35 (0.62/0.98)	**58.50/55.74** (0.38/0.96)	55.83/52.16 (0.31/1.78)	56.99/53.13 (0.42/0.89)	57.95/54.36 (0.68/1.20)	50.02/46.83 (0.92/2.73)	51.23/47.23 (1.02/3.01)	50.23/48.11 (1.03/2.50)
WineQuality	5318	11	7	53.95/53.29 (0.40/0.57)	**55.52**/53.29 (0.19/0.81)	52.96/52.88 (0.39/0.45)	53.09/52.84 (0.33/0.46)	53.35/53.08 (0.41/0.61)	55.21/55.60 (0.37/1.24)	54.96/**55.02** (0.32/0.98)	55.42/54.32 (1.21/0.42)
SteelOthers	1941	27	2	73.30/73.57 (1.09/0.44)	77.77/73.57 (0.16/1.33)	73.17/73.37 (0.82/1.06)	74.38/74.50 (1.55/1.02)	76.10/75.32 (0.96/0.88)	76.49/**75.42** (0.81/0.74)	**77.81**/74.38 (0.32/0.56)	74.32/72.5 (0.34/1.83)
DrgCnsAlc	1885	12	7	69.35/69.44 (0.43/1.91)	70.62/69.55 (0.50/1.85)	69.24/69.71 (0.69/2.12)	69.44/69.34 (0.55/2.02)	70.03/**70.08** (0.78/1.74)	72.44/69.18 (0.34/1.45)	72.88/68.96 (0.27/1.42)	**73.21**/69.43 (0.43/1.58)
DrgCnsImp	1885	12	7	41.22/38.57 (0.52/1.38)	43.83/38.20 (0.43/3.13)	40.69/39.47 (0.67/1.87)	41.07/38.99 (0.69/1.61)	41.27/38.89 (0.73/1.59)	46.18/**42.65** (2.43/4.38)	**46.25**/42.60 (3.18/4.37)	46.06/40.80 (4.01/2.32)
DrgCnsSS	1885	12	7	52.06/51.03 (0.53/0.024)	54.23/51.57 (0.38/2.22)	51.54/51.88 (0.36/2.75)	51.80/51.41 (0.22/2.39)	51.80/51.41 (0.22/2.39)	55.48/51.57 (0.50/1.60)	55.82/51.36 (0.60/1.33)	**56.21/52.42** (0.32/1.24)
EstObLvl	2087	16	7	65.07/63.16 (0.12/2.07)	**72.51/70.05** (0.28/1.54)	61.25/60.62 (0.35/1.40)	67.31/67.22 (0.31/1.33)	69.50/68.66 (0.23/1.4)	51.70/48.95 (1.61/3.17)	49.61/46.89 (1.75/2.51)	49.64/49.22 (1.46/2.84)
AiDS	2139	23	2	89.23/88.55 (0.15/0.74)	**90.24/89.11** (0.20/0.86)	89.23/88.55 (0.15/0.74)	89.36/88.60 (0.27/0.80)	89.49/88.64 (0.23/0.84)	86.99/85.93 (1.87/2.23)	86.99/85.65 (2.19/2.51)	87.02/85.23 (1.32/2.44)
AucVer	2043	7	2	92.08/90.56 (0.65/2.22)	**94.98/93.55** (0.40/1.50)	91.91/90.27 (0.64/2.16)	92.83/91.00 (0.52/2.08)	93.05/91.30 (0.46/2.21)	90.00/88.36 (0.28/1.53)	90.11/88.66 (0.48/1.38)	89.92/88.31 (0.35/1.65)
Ai4iMF	10000	6	2	97.41/97.23 (0.05/0.48)	97.84/97.27 (0.06/0.22)	97.22/97.06 (0.28/0.39)	97.49/97.19 (0.26/0.32)	97.56/97.26 (0.19/0.38)	**97.92/98.01** (0.11/0.17)	97.88/97.92 (0.12/0.17)	97.82/97.85 (0.17/0.09)
VoicePath	704	2	2	97.16/95.04 (0.48/1.85)	98.15/96.03 (0.43/1.83)	97.41/95.04 (0.53/2.24)	97.58/95.18 (0.47/2.26)	97.73/95.04 (0.51/2.33)	98.12/97.89 (0.27/1.42)	98.30/97.89 (0.27/1.00)	**98.40/98.01** (0.25/1.27)
WaveForm	5000	21	3	73.29/70.44 (0.32/0.41)	**76.53/73.26** (0.16/1.03)	71.37/68.84 (0.45/1.17)	72.36/69.84 (0.62/1.16)	72.69/69.96 (0.79/1.17)	64.37/62.70 (0.82/1.11)	64.58/62.70 (3.71/1.82)	66.12/62.70 (2.52/3.64)

Table 10: (Part II: K in 4, 5, 6) Five-fold cross-validation results on the UCI dataset. We compare the performance of our HDT algorithm, with K (number of splitting rules) ranging from 4 to 6, trained using $sodtWSH$—against approximate methods:size- and depth-constrained CART algorithms (CART-size and CART-depth), as well as the state-of-the-art optimal axis-parallel decision tree algorithm, ConTree. The depth of the CART-depth and ConTree algorithms are fixed at 3. Results are reported as mean accuracy on the training and test sets in the format *Training Error / Test Error (Standard Deviation: Train / Test)*. The best-performing algorithm in each row is shown in **bold**.