Anytime Optimal Decision Tree Learning with Continuous Features

Harold Kiossou[1 [0000−0001−6972−9885]](✉), Pierre Schaus[1 [0000−0002−3153−8941]], and Siegfried Nijssen[1,2 [0000−0003−2678−1266]]

[1] ICTEAM, UCLouvain, Belgium, {first.last}@uclouvain.be
[2] DTAI, KU LEUVEN, Belgium, siegfried.nijssen@kuleuven.be

Abstract. In recent years, significant progress has been made on algorithms for learning optimal decision trees, primarily in the context of binary features. Extending these methods to continuous features remains substantially more challenging due to the large number of potential splits for each feature. Recently, an elegant exact algorithm was proposed for learning optimal decision trees with continuous features; however, its practical applicability remains limited to shallow depths (typically 3 or 4). It relies on a depth-first search strategy that fully optimizes the left subtree of each split before exploring the corresponding right subtree. While effective in finding optimal solutions given sufficient time, this strategy can lead to poor anytime behavior: when interrupted early, the best-found tree is often highly unbalanced and suboptimal. In such cases, purely greedy methods such as C4.5 may, paradoxically, yield better solutions. To address this limitation, we propose an anytime yet complete approach that leverages limited discrepancy search, distributing the computational effort more evenly across the entire tree structure, and thus ensuring that a high-quality decision tree is available at any interruption point. Experimental results show that our approach outperforms the existing one in terms of anytime performance.

Keywords: Decision Trees · Anytime Search · Discrepancy Search.

1 Introduction

Decision trees are widely used in machine learning due to their simplicity and interpretability, with applications in domains such as healthcare, finance, and education. Learning decision trees is typically performed using greedy algorithms, such as CART [7] and C4.5 [26], which construct trees in a top-down manner by selecting a split at each node according to a heuristic criterion. While these greedy approaches are highly scalable, they often produce trees that are less accurate than their optimal counterparts [20]. On the other hand, learning an optimal decision tree, i.e., finding a tree that minimizes the training classification error subject to constraints such as a maximum depth, is NP-hard [18]. The difficulty stems from the combinatorial search space, which grows rapidly with the tree depth and with the number of candidate splits to be considered. Despite this computational challenge, recent advances in optimization techniques

have made it increasingly feasible to learn optimal decision trees in practice on small- to medium-sized datasets. These algorithms leverage combinatorial optimization techniques from mixed-integer linear programming (MILP) [1, 5], constraint programming [27], and SAT [23], but they struggle to scale with dataset size. Dynamic programming (DP) and branch-and-bound (BnB) approaches [2, 9] significantly improve scalability, but typically assume binary features. As a result, numeric attributes must either be discretized beforehand, which can restrict the set of candidate trees considered, or represented by introducing one binary feature for each candidate threshold. The latter preserves the relevant threshold candidates but substantially increases the effective number of binary features, which can severely harm scalability.

To address this limitation, Quant-BnB was proposed [21]; this is a specialized algorithm that learns optimal decision trees directly from numeric data. It considers splits at selected quantiles of the feature distribution and uses the resulting solutions to prune other parts of the search space. While Quant-BnB significantly outperforms previous approaches on numeric features, it does not scale beyond depth 3. ConTree [8] is a subsequent algorithm that combines dynamic programming and branch-and-bound techniques with novel bounding strategies and a specialized solver for depth-2 optimal decision trees. This enables learning up to depth 4 optimal decision trees on medium-sized numeric datasets within a reasonable time.

While effective at proving optimality, these methods are not *anytime algorithms*. An *anytime algorithm* rapidly produces good-quality solutions, and can be interrupted at any time to return its best solution found so far. Given sufficient time, it eventually finds and certifies an optimal solution. Existing algorithms focus on proving optimality and prioritize bound tightening over early solution quality. Approaches such as DL8.5 [2] or ConTree explore the search space in a depth-first fashion, which often causes it to become stuck in unpromising regions, as illustrated in Figure 1. As a result, they may return poor trees when stopped early. Greedy methods like C4.5 provide quick results but lack the capacity to improve or guarantee optimality over time. Neither approach provides the benefits of a true anytime algorithm, which should quickly produce a good initial solution and then continuously improve it as time permits, eventually proving optimality.

To improve the anytime performance or scalability of exact methods, two notable works have been proposed. LDS-DL8.5 [16] integrates limited discrepancy search (LDS) into DL8.5, resulting in an algorithm that is both anytime and complete. This approach was further generalized using an anytime beam search in [14]. The Blossom algorithm [10] follows a fundamentally different search strategy. It uses a depth-first approach that expands decision tree nodes level by level, offering improved anytime behavior by avoiding the potential for highly unbalanced trees when interrupted early, as in DL8.5. Blossom is guaranteed to find the optimal tree given sufficient time.

Top-k-DL8.5 [6] modifies DL8.5 by restricting the candidate features at each node to the Top-k according to a ranking heuristic. This is a compromise between

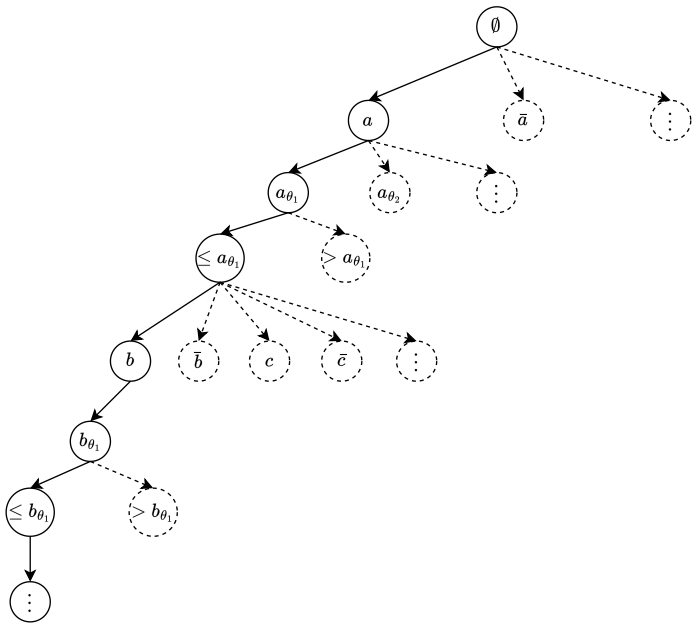

Fig. 1. ConTree explores the leftmost branches first, often leading to poor anytime performance when interrupted early.

C4.5 and DL8.5: faster and more scalable, but unable to guarantee convergence to the optimal tree.

Another line of research focuses on learning better, though not necessarily optimal, trees using less greedy algorithms [3, 6, 15, 17]. These approaches either restrict the set of candidate features considered at each node, rely on lookahead strategies to make more informed splitting decisions, or use greedy decision trees as a baseline to reduce the search space. While such methods can improve upon purely greedy trees while remaining scalable, they do not guarantee optimality and are typically not designed as anytime algorithms.

In this work, we introduce CA-ConTree, a novel algorithm for learning decision trees on continuous features that is **anytime**. Limited discrepancy search allows CA-ConTree to prioritize promising regions of the search space early and progressively broaden exploration over time. Our experimental results show that CA-ConTree exhibits strong anytime behavior, outperforming ConTree on medium-sized datasets at depths 4 and 5, and produces trees that generalize better under time constraints.

This paper is organized as follows. Section 2 reviews related work on optimal decision tree learning. Section 3 introduces the necessary background and notation. Section 4 presents CA-ConTree and details its search procedure. Finally, Section 5 reports the experimental evaluation.

2 Related Work

Greedy approaches Learning decision trees has historically relied on greedy algorithms such as CART [7] and C4.5 [26], which optimize local splitting criteria at each node. While these approaches scale well, they are prone to suboptimal solutions due to their myopic nature. In particular, greedy induction tends to produce trees that are larger than the optimal tree on average [22], or, when constrained to a fixed depth, yields lower out-of-sample accuracy than optimal trees under the same size limit [20]. As a result, recent research has shifted toward Optimal Decision Trees (ODTs) that leverage combinatorial optimization techniques. These include Mixed-Integer Programming formulations [5] and logic-based paradigms such as SAT [23], MaxSAT [12], and Constraint Programming (CP) [27]. However, because finding a minimum-error decision tree under constraints such as a maximum depth is NP-hard, these formulations must search over a combinatorial space of tree structures and split assignments. This search space grows rapidly with the depth, the number of features, and the number of candidate splits, which limits scalability on large datasets and deeper depths.

Dynamic programming approaches Nijssen and Fromont [24, 25] introduced DL8, an early dynamic programming approach for optimal decision tree learning on binary features. This was later improved by [2] with DL8.5, which incorporates branch-and-bound and enhanced caching techniques to significantly reduce the search space. Other works focused on deriving stronger bounds: [13] and [19] proposed new lower bounds, including a subproblem similarity bound, to prune redundant computations. More recently, [9] introduced a specialized subroutine for depth-two trees and additional constraints to limit the number of branching nodes, further improving efficiency.

Continuous features Specialized approaches for learning optimal decision trees (ODTs) directly from continuous data are relatively recent. The first such method, Quant-BnB (Mazumder, Meng, and Wang, 2022), tries to learn optimal decision trees on continuous data by splitting on quantiles of the feature distribution and using the results to bound other parts of the search space. Although it can handle much larger datasets than MIP or SAT-based approaches, it struggles to scale beyond trees of depth three. More recently, [8] proposed ConTree, which leverages the Similarity Lower Bound introduced in OSDT and GOSDT [13, 19] to implement three types of pruning and lower bounds. This allows ConTree to reduce both the computational cost and the search space, enabling it to scale better than Quant-BnB and learn ODTs on medium-sized datasets with a maximum depth of 4.

Anytime approaches Dynamic programming approaches have one notable limitation: if interrupted, they typically produce an incomplete or unbalanced decision tree, which can be of lower quality than one constructed by greedy heuristics. To address this and improve anytime performance, recent work has explored strategies for generating good intermediate solutions during the search. The first such

approach is [16], which introduced LDS-DL8.5, applying iterative Limited Discrepancy Search (LDS) to prioritize solutions that are close to a heuristic baseline tree first.

Similarly, the Blossom algorithm [10] employs a different search strategy to improve anytime behavior. While it also uses depth-first search, Blossom proceeds layer by layer, always expanding the non-expanded node closest to the root. As with LDS-DL8.5, the first solution found corresponds to the tree produced by a purely greedy strategy.

3 Background

Learning an optimal decision tree (ODT) is performed on a dataset $\mathcal{D}$ consisting of $n = |\mathcal{D}|$ observations (x, y), where in this work features are numerical and hence $x \in \mathbb{R}^p$ is the feature vector and $y \in \mathcal{Y}$ is the corresponding class label. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_p\}$ denote the set of features. For an observation x and a feature $f \in \mathcal{F}$, we write x_f for the value of feature f in x.

For a given feature f, let $\mathcal{U}^f = (\mathcal{U}^f_1, \ldots, \mathcal{U}^f_{|\mathcal{U}^f|})$ denote the sorted sequence of unique values taken by feature f in $\mathcal{D}$, with $\mathcal{U}^f_1 < \cdots < \mathcal{U}^f_{|\mathcal{U}^f|}$. We define the set of candidate split thresholds for feature f as

$$S^f = \left\{ \frac{\mathcal{U}^f_i + \mathcal{U}^f_{i+1}}{2} \;\middle|\; i = 1, \ldots, |\mathcal{U}^f| - 1 \right\}, \tag{1}$$

which contains the midpoints between consecutive unique values of f.

Given a threshold $\tau \in S^f$, let $\mathcal{D}(f \leq \tau)$ denote the subset of observations $(x, y) \in \mathcal{D}$ such that $x_f \leq \tau$.

Let $\mathcal{T}(\mathcal{D}, d)$ denote the set of decision trees over $\mathcal{D}$ with maximum depth d, using candidate thresholds S^f for each feature $f \in \mathcal{F}$. The optimal classification tree of depth at most d is then defined as

$$t^\star_d = \arg \min_{t \in \mathcal{T}(\mathcal{D}, d)} \sum_{(x,y) \in \mathcal{D}} \mathbb{I}[t(x) \neq y]\,. \tag{2}$$

3.1 ConTree

ConTree (CT) learns ODTs by recursively performing splits. Subproblems below splits are identified by subsets of the dataset $\mathcal{D}$ and the remaining depth limit d. This can be expressed with the following recurrence equation for determining the lowest misclassification score that a tree of depth d can have on a dataset $\mathcal{D}$:

$$CT(\mathcal{D}, d) = \begin{cases} \min\limits_{\hat{y} \in \mathcal{Y}} \sum\limits_{(x,y) \in \mathcal{D}} \mathbb{I}[\hat{y} \neq y], & \text{if } d = 0 \\ \min\limits_{f \in \mathcal{F}, \tau \in S^f} [CT(\mathcal{D}_{f \leq \tau}, d-1) & \\ \qquad + CT(\mathcal{D}_{f > \tau}, d-1)], & \text{if } d > 0. \end{cases} \tag{3}$$

At each internal node, the algorithm considers pairs (f, τ), where $f \in \mathcal{F}$ is a feature and $\tau \in S^f$ is one of its candidate thresholds. Each candidate split partitions the current dataset into the left subset $\mathcal{D}(f \leq \tau)$ and the right subset $\mathcal{D}(f > \tau)$. The selected split is the one that minimizes the sum of the optimal misclassification scores of the two resulting subproblems, subject to the remaining depth.

For a fixed feature, candidate thresholds are explored using a binary interval-splitting order : the midpoint threshold of the current interval is considered first, then the same procedure is applied recursively to the left and right subintervals. This ordering progressively refines the search over the thresholds, but it is not guided by a data dependent impurity heuristic.

4 CA-ConTree

In this work, we propose the Complete Anytime Continuous Tree (CA-ConTree) algorithm, which leverages Limited Discrepancy Search (LDS) [11] together with heuristic orderings of features and split thresholds to obtain an anytime algorithm for optimal decision tree learning with continuous features. LDS explores solutions in increasing order of deviation from a heuristic, based on the assumption that high-quality solutions often differ from the heuristic-guided solution by only a small number of decisions.

4.1 Limited Discrepancy Search

LDS is based on the idea that high-quality solutions are often close to the solution suggested by a heuristic. Rather than following a standard depth-first search order, LDS first explores decisions that agree with the heuristic and then gradually allows deviations from this ordering. Each deviation from the heuristic incurs a discrepancy cost. A discrepancy budget limits the total deviation allowed during a search iteration.

In CA-ConTree, LDS is implemented as an iterative procedure. At each iteration, the algorithm searches for the best tree that can be reached under the current discrepancy budgets. These budgets are then increased, allowing the next iteration to explore a larger part of the search space. Therefore, small budgets provide strong anytime behavior by focusing on promising decisions first, while larger budgets progressively recover completeness.

4.2 Feature and Split Discrepancy Budgets

The main difficulty in applying LDS to continuous features is that each feature can induce many candidate thresholds. A single discrepancy budget would not distinguish between deviating from the best feature and deviating from the best threshold of a fixed feature. To obtain finer control over the search, CA-ConTree uses two discrepancy budgets.

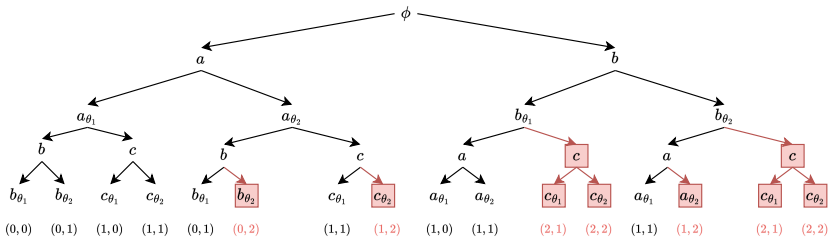

Fig. 2. Representation of the search tree for decision tree learning on numeric data with a feature discrepancy budget of 1 and a split discrepancy budget of 1. For each feature f, candidate split decisions are represented as f_{θ_i}, where θ_i denotes the index of a possible threshold in the ordered split set S^f. Nodes shown in red correspond to decisions that are not explored due to the imposed discrepancy budgets.

The first budget, denoted *featDisc*, controls deviations from the heuristic ordering of features. The second budget, denoted *splitDisc*, controls deviations from the heuristic ordering of thresholds for a selected feature. In our implementation, both features and thresholds are ordered using the Gini score.

More precisely, suppose that, at a given node, the features are sorted by the heuristic. Selecting the first-ranked feature incurs no feature discrepancy, selecting the second-ranked feature incurs one unit of feature discrepancy, and, in general, selecting the feature at rank i incurs $i-1$ units of feature discrepancy. The same principle is applied to the thresholds of a selected feature: selecting the threshold ranked j incurs $j-1$ units of split discrepancy.

Along a root-to-leaf branch, the cumulative feature discrepancy must not exceed *featDisc*, and the cumulative split discrepancy must not exceed *splitDisc*. Thus, the budget pair (*featDisc*, *splitDisc*) defines the part of the search space explored during one LDS iteration. Small budgets restrict the search to choices close to the heuristic ordering, whereas larger budgets allow increasingly distant feature and threshold choices to be considered.

Figure 2 illustrates this principle for a search tree with maximum discrepancy budgets $(1, 1)$. The leftmost branch corresponds to the best choices according to the heuristic and therefore has discrepancy cost $(0, 0)$. Choosing a lower-ranked feature or threshold increases the corresponding discrepancy. Nodes whose cumulative discrepancy exceeds the current budgets are not explored during that iteration. However, they may be explored later when the budgets are increased.

4.3 Search Procedure

Algorithms 1 and 2 present the main structure of CA-ConTree. Algorithm 1 describes the global anytime loop. The outer loop repeatedly invokes the recursive procedure CT while progressively increasing the discrepancy budgets (Line 6) and tightening the global upper bound $\mathcal{UB}$ whenever a better incumbent is found (Line 5).

Algorithm 1: CA-ConTree($\mathcal{D}, d, \mathcal{UB}$)

```
1  θ_opt ← min_{ŷ∈Y} Σ_{(x,y)∈D} 𝕀(ŷ ≠ y)
2  featDisc ← 0, splitDisc ← 0
3  while budgetRemains(featDisc, splitDisc) do
4  |  θ_opt ← CT(D, d, UB, featDisc, splitDisc)
5  |  UB ← min(θ_opt, UB)
6  |  (featDisc, splitDisc) ← nextBudget(featDisc, splitDisc)
7  end
8  Procedure CT(D, d, UB, featDisc, splitDisc)
9  |  if d = 0 then return min_{ŷ∈Y} Σ_{(x,y)∈D} 𝕀(ŷ ≠ y)
10 |  F_ordered ← heuristicSort(D)
11 |  for it ← 0 to |F_ordered| − 1 do
12 |  |  f ← F_ordered[it]
13 |  |  disc ← featDisc − it
14 |  |  if disc < 0 then return θ_opt
15 |  |  θ_f ← Branch(D, d, f, UB, disc, splitDisc)
16 |  |  if θ_f < θ_opt then
17 |  |  |  θ_opt ← θ_f, UB ← θ_f
18 |  |  end
19 |  end
20 |  return θ_opt
21 end
22 return θ_opt
```

The procedure CT solves a subproblem defined by a dataset $\mathcal{D}$, a remaining depth d, an upper bound $\mathcal{UB}$, and the current discrepancy budgets. If $d = 0$, the best possible tree is a leaf, and the procedure returns the misclassification score of the best constant prediction. Otherwise, CT expands features according to the current feature discrepancy budget. Features are first ordered by a heuristic. The algorithm then considers the features in this order, subtracting the feature rank from the remaining feature discrepancy budget before calling Algorithm 2. Thus, only features whose discrepancy cost is admissible under the current budget are explored.

Algorithm 2 searches over the candidate split thresholds of a fixed feature. In ConTree [8], thresholds are enumerated using a binary interval-splitting order: the algorithm first evaluates the threshold at the midpoint of the current interval of candidate split indices and then recursively applies the same procedure to the left and right subintervals. This ordering progressively refines the interval of candidate thresholds, but it is not guided by a data-dependent splitting heuristic.

CA-ConTree instead orders candidate thresholds using a heuristic that estimates their split quality (Line 2) and explores them sequentially under the current split discrepancy budget. For each threshold w, Algorithm 2 partitions the dataset into $\mathcal{D}(f \leq S_w^f)$ and $\mathcal{D}(f > S_w^f)$ (Line 11), and recursively calls CT on the resulting subproblems with the updated remaining split discrepancy bud-

Algorithm 2: Branch($\mathcal{D}, d, f, \mathcal{UB}, featDisc, splitDisc$)

```
1  θ_opt ← min_{ŷ∈Y} Σ_{(x,y)∈D} 𝕀(ŷ ≠ y)
2  splits ← heuristicSort({1, 2, ..., m})
3  pruned ← 0^|splits|
4  for it ← 0 to |splits| − 1 do
5      if pruned[it] = 0 then
6          w ← splits[it]
7          disc ← splitDisc − it
8          if disc < 0 then return θ_opt
9          if d = 2 then θ_{w,L}, θ_{w,R} ← D2Split(D, f, w)
10         else
11             D_L ← D(f ≤ S_w^f), D_R ← D(f > S_w^f)
12             θ_{w,L} ← CT(D_L, d − 1, UB, featDisc, disc)
13             θ_{w,R} ← CT(D_R, d − 1, UB − θ_{w,L}, featDisc, disc)
14         end
15         θ_w ← θ_{w,L} + θ_{w,R}
16         if θ_w < θ_opt then
17             θ_opt ← θ_w, UB ← min(UB, θ_w)
18         end
19         Δ ← max(1, θ_w − UB);
20         foreach k ∈ {w − Δ, ..., w + Δ} do
21             pruned[k] ← 1
22         end
23     end
24 end
25 return θ_opt
```

get. If the resulting split improves the current solution, the solution value and the upper bound $\mathcal{UB}$ are updated (Lines 15–18), allowing subsequent recursive calls to benefit from stronger pruning.

When the remaining depth equals two, CA-ConTree invokes the *D2Split* procedure of ConTree [8] instead of continuing the generic recursion (Line 9). This subroutine solves the depth-two subproblem exactly and therefore provides a stronger base case than a leaf-only termination.

4.4 Similarity Lower Bound and Neighborhood Pruning

To avoid evaluating all candidate thresholds, CA-ConTree uses neighborhood pruning based on the Similarity Lower Bound (SLB) [9, 13, 19] as used in ConTree. SLB compares a new dataset $\mathcal{D}_{\text{new}}$ with a previously analyzed dataset $\mathcal{D}_{\text{old}}$ to derive a lower bound on the misclassification score of the new dataset. It assumes that all observations present in $\mathcal{D}_{\text{new}}$ are classified correctly, while all observations removed from $\mathcal{D}_{\text{old}}$ are classified incorrectly. This gives the following lower bound:

$$\theta_{\mathcal{D}_{\text{new}}} \geq \theta_{\mathcal{D}_{\text{old}}} - |\mathcal{D}_{\text{old}} \setminus \mathcal{D}_{\text{new}}|, \tag{4}$$

where $\theta_{\mathcal{D}}$ denotes the minimum misclassification score achievable by a decision tree with the same depth limit on dataset $\mathcal{D}$.

After evaluating a threshold w, CA-ConTree uses the SLB to discard neighboring thresholds that cannot improve the current upper bound within the current LDS iteration. In Algorithm 2, this corresponds to the pruning step performed after computing the split value θ_w (Lines 19–22). The pruning radius depends on the gap between the value obtained for the current split and the current upper bound. This pruning is local to the current discrepancy budgets. In other words, θ_w is interpreted as the best value obtained for split w under the current budget configuration. When the budgets are increased in later iterations, the corresponding node may be revisited and previously pruned thresholds may be reconsidered. Therefore, neighborhood pruning improves efficiency within an LDS iteration without compromising the completeness of the overall search.

4.5 Budget Scheduling

The order in which discrepancy budgets are increased has a strong impact on anytime behavior. A simple strategy is to increase both budgets linearly. However, because continuous features may induce many thresholds, this can spend too much time expanding threshold choices without sufficiently diversifying the explored tree structures. Another possibility is to increase the budgets exponentially. This covers larger portions of the search space more quickly, but the large jumps may cause the search to spend too much time in expensive regions before improving the incumbent.

In this work, we use a diagonal sweep over the two-dimensional budget space. Budget pairs are ordered by the sum of their feature and split discrepancies. The algorithm therefore explores [(0,0), (1,0), (0,1), (2,0), (1,1), (0,2), ...] This schedule gradually increases the total allowed discrepancy while alternating between feature and threshold deviations. It therefore balances the exploration of alternative features and alternative thresholds.

Figure 3 compares this strategy with linear and exponential budget schedules on the Avila dataset with maximum depth 4 and a timeout of 600 seconds. The diagonal sweep strategy improves the current solution more rapidly and achieves better anytime behavior in this representative case. We therefore use diagonal sweep in all subsequent experiments.

4.6 Caching

To avoid redundant computations across search iterations, we employ the same dataset caching strategy as in [9]. This cache is used in particular for calls to the *D2Split* subroutine. As discrepancy budgets increase, identical subproblems may be revisited multiple times; retrieving cached optimal solutions avoids repeating these computations. Moreover, within a single search iteration, a cached solution can be reused whenever it was computed under discrepancy budgets that are at least as large as the current remaining budgets.

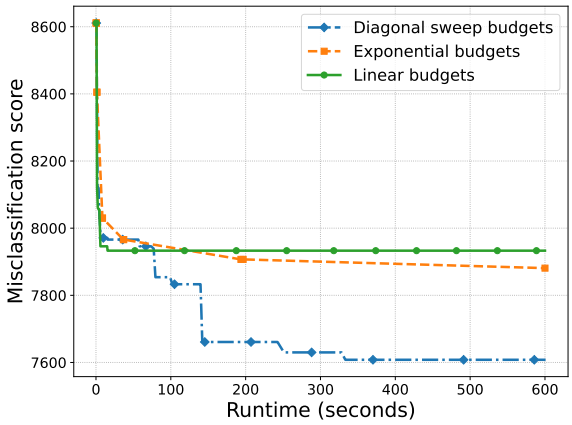

Fig. 3. Comparison of discrepancy budget scheduling strategies on a representative dataset (depth 4).

5 Results

We conducted a series of experiments to evaluate the performance of CA-ConTree. First, we analyze the anytime behavior of CA-ConTree relative to optimal and greedy decision-tree baselines. We then evaluate its ability to generalize. The datasets and implementation used in our experiments are publicly available at https://github.com/haroldks/contree-rs. All experiments were performed on an Intel Xeon Platinum 8160 CPU with 320 GB of RAM, running Rocky Linux 8.4. The algorithms were evaluated on the 16 datasets from the UCI Machine Learning Repository previously used in [8]. Table 1 presents the main characteristics of these datasets, including the number of instances, features, and classes.

We distinguish between two types of baselines. The first group consists of optimal decision tree methods, whose objective is to minimize the training misclassification score under a maximum-depth constraint. Some of these methods require binary input features. For numerical datasets, they are therefore applied after transforming numerical attributes into binary threshold features. This comparison evaluates how existing optimal decision tree solvers behave on continuous data when using their required input representation. In contrast, CA-ConTree and ConTree operate directly on numerical attributes and consider candidate thresholds during the search.

The second group consists of greedy decision tree learners. In particular, we include C4.5 with the same maximum-depth limit as a practical reference. This allows us to compare models at a similar level of interpretability.

We compare CA-ConTree to the other algorithms using the *average primal integral*, as introduced in [4], to measure the anytime behavior of optimization solvers. The primal integral measures the progress of an algorithm's primal-bound convergence toward the optimal (or best-known) solution over the entire solving time. It is based on the *primal function* $p(t)$, which represents the gap

Table 1. Datasets used in the experiments.

Dataset	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{F}\rvert$	$\lvert\mathcal{Y}\rvert$
Avila	10430	10	12
Bank	1097	4	2
Bean	10888	16	7
Bidding	5056	9	2
Eeg	11984	14	2
Fault	1552	27	7
Htru	14318	8	2
Magic	15216	10	2
Occupancy	8143	5	2
Page	4378	10	5
Raisin	720	7	2
Rice	3048	7	2
Room	8103	16	4
Segment	1848	18	7
Skin	196045	3	3
Wilt	4339	5	5

between the current solution $x(t)$ at time t and the optimal or best known solution x_{ref}.

For the primal integral computation, the reference value x_{ref} corresponds to the best objective value obtained by any algorithm within the 600-second time limit. This value is used as a best-known reference for measuring anytime progress.

The *primal gap* of a solution $x(t)$ is defined as

$$\gamma(x(t)) = \frac{|x(t) - x_{\text{ref}}|}{|x(t)|}.$$

The function $p(t)$ equals 1 if no feasible solution has been found by time t, and $\gamma(x(t))$ otherwise. The function $p(t)$ is a step function that changes whenever a new feasible solution is found. It is monotonically decreasing and becomes zero once the reference value is reached. The primal integral $P(T)$ is defined as the integral of the primal gap function $p(t)$ over the time horizon T:

$$P(T) = \int_0^T p(t)\,dt = \sum_{i=1}^{n} p(t_{i-1}) \cdot (t_i - t_{i-1}),$$

where each t_i denotes a time point at which a new incumbent solution is found. The primal integral encourages the early discovery of high-quality solutions. If a better solution is found at the same time, $P(t_{\max})$ decreases. Similarly, if the same solution is found earlier, $P(t_{\max})$ also decreases. The ratio $P(t_{\max})/t_{\max}$ represents the average primal gap during the search process. A smaller value indicates better expected incumbent quality if the algorithm is interrupted at an arbitrary point in time.

Table 2. Average primal integral on depth 4

		Runtime (s)						
Approach	Sub	5	15	30	60	120	300	600
CA-ConTree	–	**33.5**	**26.5**	**23.1**	**20.4**	**18.3**	**16.3**	**14.6**
ConTree-Gini	–	89.1	85.4	77.4	71.8	67.6	60.0	50.5
ConTree	–	93.8	89.0	83.7	76.8	70.3	62.1	54.2
C4.5	–	40.8	40.8	40.8	40.8	40.8	40.8	40.8
LDS-DL8.5	–	53.0	52.5	52.3	52.3	52.3	52.3	52.2

Tables 2 and 3 report the evolution of the average primal integral across various time budgets (from 5 to 600 seconds) for tree depths 4 and 5. Depth 4 was used in the original ConTree evaluation and provides a suitable setting for comparing CA-ConTree with ConTree. We also evaluate depth 5 to study how the methods behave on more complex problems, where the search space is substantially larger and proving optimality becomes more difficult. We compare CA-ConTree with ConTree using and without the Gini heuristic, as well as with the greedy C4.5 baseline and LDS-DL8.5. Since LDS-DL8.5 only supports binary data, the datasets were binarised beforehand using 10 quantile-based bins, which significantly increases the number of features.

Table 3. Average primal integral on depth 5

		Runtime (s)						
Approach	Sub	5	15	30	60	120	300	600
CA-ConTree	–	**40.2**	**33.4**	**31.2**	**29.5**	**28.3**	**26.5**	**24.8**
ConTree-Gini	–	90.1	89.2	85.2	83.2	82.2	78.4	76.7
ConTree	–	93.8	93.8	93.8	93.7	92.7	83.9	79.8
C4.5	–	50.2	50.2	50.2	50.2	50.2	50.2	50.2
LDS-DL8.5	–	59.4	58.7	58.3	57.9	57.5	57.2	57.1

Across both depths, CA-ConTree consistently achieves the lowest primal integral for all time budgets, highlighting its strong anytime behavior. In contrast, ConTree and ConTree-Gini improve more slowly and remain substantially behind, while C4.5 yields a constant primal integral due to its non-anytime, greedy nature. The gap between CA-ConTree and the other complete approaches persists even at larger time budgets, indicating that the LDS strategy enables faster discovery of high-quality solutions throughout the search. The primal integral values observed for ConTree and ConTree-Gini indicate a weak anytime behavior. Although these approaches are complete and may eventually reach high-

quality or optimal solutions, they tend to maintain poor incumbents for a large portion of the runtime. As a result, improvements in solution quality occur late in the search and contribute only marginally to reducing the accumulated primal gap. This effect is only partially mitigated by the Gini heuristic. C4.5 exhibits constant primal integral values, as it immediately produces a greedy solution and does not refine it further. While C4.5 lacks optimality guarantees, its early availability of a reasonable solution allows it to outperform ConTree in the primal integral for small and medium time budgets.

LDS-DL8.5 shows better primal integral values than ConTree and ConTree-Gini, indicating that it is able to obtain stronger incumbent solutions earlier during the search. However, its performance remains behind CA-ConTree. Moreover, the binarisation required by LDS-DL8.5 substantially increases the number of features, thereby significantly enlarging the search space. This feature explosion tends to degrade the overall quality of the resulting trees, as the binary representation limits the expressiveness of splits compared to continuous thresholds. As a result, despite its stronger anytime behavior than ConTree and ConTree-Gini, the solutions produced by LDS-DL8.5 are generally lower quality than those obtained by the greedy C4.5 baseline.

Table 4. Average test set accuracy on depth 5

Dataset	Train. Size	C4.5	ConTree	CA-ConTree	LDS-DL8.5
skin	196045	0.984 ± 0.000	**0.994 ± 0.002**	**0.994 ± 0.000**	0.973 ± 0.001
avila	16693	0.620 ± 0.010	0.616 ± 0.005	**0.665 ± 0.006**	0.629 ± 0.007
occupancy	16448	0.989 ± 0.003	**0.992 ± 0.002**	0.991 ± 0.002	0.979 ± 0.001
magic	15216	0.820 ± 0.004	0.732 ± 0.007	**0.846 ± 0.004**	0.806 ± 0.003
htru	14318	**0.978 ± 0.002**	0.935 ± 0.033	0.977 ± 0.002	0.977 ± 0.002
eeg	11984	0.698 ± 0.009	0.714 ± 0.003	**0.765 ± 0.002**	0.702 ± 0.006
bean	10888	0.891 ± 0.006	0.722 ± 0.009	**0.903 ± 0.006**	0.711 ± 0.007
room	8103	0.995 ± 0.001	0.996 ± 0.001	**0.997 ± 0.001**	0.995 ± 0.002
bidding	5056	0.996 ± 0.002	0.995 ± 0.003	**0.997 ± 0.001**	0.977 ± 0.003
page	4378	0.964 ± 0.003	0.963 ± 0.005	**0.970 ± 0.003**	0.965 ± 0.007
wilt	3871	0.978 ± 0.006	0.980 ± 0.002	**0.982 ± 0.004**	0.970 ± 0.004
rice	3048	**0.919 ± 0.013**	0.917 ± 0.010	0.916 ± 0.009	0.915 ± 0.005
segment	1848	0.914 ± 0.009	**0.964 ± 0.009**	0.944 ± 0.008	0.593 ± 0.025
fault	1552	0.667 ± 0.018	0.677 ± 0.013	**0.691 ± 0.007**	0.689 ± 0.012
bank	1097	0.978 ± 0.010	**0.986 ± 0.007**	0.985 ± 0.005	0.929 ± 0.011
raisin	720	**0.857 ± 0.023**	0.829 ± 0.021	0.831 ± 0.013	0.849 ± 0.022

Next, we evaluate the test-set performance of ConTree and CA-ConTree against the C4.5 and LDS-DL8.5 baseline algorithms using 5-fold cross-validation on the same datasets, with a timeout of 600 seconds. Tables 4 and 5 report the results for tree depths 5 and 6, respectively. Overall, CA-ConTree achieves the best or near-best test accuracy on a majority of datasets, frequently outperforming both C4.5 and the baseline ConTree. In particular, ConTree often fails to surpass the C4.5 heuristic on several datasets, such as MAGIC, BEAN, and EEG, whereas CA-ConTree consistently improves upon the baseline ConTree and frequently yields the highest accuracy. These improvements are especially visible on datasets such as AVILA, MAGIC, BEAN, FAULT, and EEG, where CA-ConTree

achieves noticeable gains at both depths. On the other hand, while LDS-DL8.5 performs well in some instances, its accuracy is generally lower than CA-ConTree on most datasets.

Table 5. Average test set accuracy on depth 6

Dataset	Train. Size	C4.5	ConTree	CA-ConTree	LDS-DL8.5
skin	196045	0.989 ± 0.000	0.991 ± 0.000	**0.997 ± 0.000**	0.984 ± 0.001
avila	16693	0.656 ± 0.010	0.627 ± 0.008	**0.709 ± 0.005**	0.674 ± 0.007
occupancy	16448	0.989 ± 0.003	**0.992 ± 0.002**	0.991 ± 0.002	0.985 ± 0.000
magic	15216	0.841 ± 0.006	0.716 ± 0.006	**0.854 ± 0.006**	0.813 ± 0.004
htru	14318	**0.977 ± 0.002**	0.906 ± 0.001	0.976 ± 0.002	**0.977 ± 0.002**
eeg	11984	0.722 ± 0.016	0.711 ± 0.010	**0.785 ± 0.003**	0.716 ± 0.007
bean	10888	0.897 ± 0.005	0.582 ± 0.009	**0.908 ± 0.004**	0.769 ± 0.007
room	8103	**0.996 ± 0.001**	**0.996 ± 0.001**	0.995 ± 0.001	**0.996 ± 0.001**
bidding	5056	0.995 ± 0.001	0.995 ± 0.001	**0.997 ± 0.001**	0.976 ± 0.001
page	4378	**0.967 ± 0.004**	0.957 ± 0.006	**0.967 ± 0.006**	0.962 ± 0.005
wilt	3871	**0.980 ± 0.002**	0.978 ± 0.005	0.977 ± 0.003	0.976 ± 0.004
rice	3048	**0.915 ± 0.009**	0.890 ± 0.021	0.900 ± 0.011	0.914 ± 0.004
segment	1848	0.937 ± 0.008	0.739 ± 0.010	**0.947 ± 0.012**	0.687 ± 0.026
fault	1552	0.673 ± 0.023	0.626 ± 0.033	**0.734 ± 0.018**	0.716 ± 0.004
bank	1097	0.983 ± 0.005	**0.988 ± 0.006**	0.980 ± 0.010	0.962 ± 0.011
raisin	720	**0.841 ± 0.019**	0.806 ± 0.012	0.838 ± 0.021	0.829 ± 0.028

The benefits of CA-ConTree are most pronounced on medium-sized datasets, where its ability to discover strong solutions early and refine them over time leads to consistent improvements in final performance. On a small number of datasets, C4.5 remains competitive or slightly superior, particularly at larger depths; however, these differences are generally modest. Overall, these results indicate that integrating limited discrepancy search does not degrade generalization performance and instead provides a more robust framework for learning high-quality trees compared to both greedy heuristics and standard exact baselines.

6 Conclusion

We introduced CA-ConTree, a novel algorithm for learning optimal decision trees on continuous features with strong anytime performance. By integrating limited discrepancy search and carefully controlling the exploration of both feature and split discrepancies, CA-ConTree can discover high-quality solutions early and improve upon them. Experimental results on a diverse set of benchmark datasets demonstrated that CA-ConTree consistently outperforms existing exact methods in time-limited settings, particularly on medium-sized datasets and for tree depths where standard approaches struggle. Compared to ConTree, CA-ConTree achieves substantially better anytime behavior without sacrificing generalization performance, and often surpasses both greedy baselines and exact solvers within the same computational budget. While ConTree remains more effective when the primary objective is to prove optimality on shallow trees, CA-ConTree provides a more practical alternative in scenarios where high-quality solutions are

required quickly. As future work, it could be interesting to combine the two approaches and dynamically switch strategies, for example, after a certain amount of time. Moreover, our current work focuses on classification, but an interesting challenge is how to obtain similar results for other tasks, such as regression.

References

1. Aghaei, S., Gómez, A., Vayanos, P.: Strong optimal classification trees. Operations Research **73**(4), 2223–2241 (2025)
2. Aglin, G., Nijssen, S., Schaus, P.: Learning optimal decision trees using caching branch-and-bound search. In: Proceedings of AAAI. vol. 34, pp. 3146–3153 (2020)
3. Babbar, V., McTavish, H., Rudin, C., Seltzer, M.: Near optimal decision trees in a split second. arXiv preprint arXiv:2502.15988 (2025)
4. Berthold, T.: Measuring the impact of primal heuristics. Operations Research Letters **41**(6), 611–614 (2013). https://doi.org/https://doi.org/10.1016/j.orl.2013.08.007, https://www.sciencedirect.com/science/article/pii/S0167637713001181
5. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning **106**, 1039–1082 (2017)
6. Blanc, G., Lange, J., Pabbaraju, C., Sullivan, C., Tan, L., Tiwari, M.: Harnessing the power of choices in decision tree learning. In: Advances in Neural Information Processing Systems. vol. 36 (2024)
7. Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and regression trees, vol. 37. Wadsworth Int. Group (1984)
8. Briţa, C.E., van der Linden, J.G., Demirović, E.: Optimal classification trees for continuous feature data using dynamic programming with branch-and-bound. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 39, pp. 11131–11139 (2025)
9. Demirović, E., Lukina, A., Hebrard, E., Chan, J., Bailey, J., Leckie, C., Ramamohanarao, K., Stuckey, P.J.: Murtree: Optimal decision trees via dynamic programming and search. Journal of Machine Learning Research **23**(26), 1–47 (2022)
10. Demirović, E., Hebrard, E., Jean, L.: Blossom: an anytime algorithm for computing optimal decision trees. In: International Conference on Machine Learning. pp. 7533–7562 (2023)
11. Harvey, W., Ginsberg, M.: Limited discrepancy search. In: IJCAI (1). pp. 607–615 (1995)
12. Hu, H., Siala, M., Hebrard, E., Huguet, M.J.: Learning optimal decision trees with maxsat and its integration in adaboost. In: IJCAI-PRICAI 2020, 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence (2020)
13. Hu, X., Rudin, C., Seltzer, M.: Optimal sparse decision trees. Advances in neural information processing systems **32** (2019)
14. Kiossou, H., Schaus, P.: A generic complete anytime beam search for optimal decision tree. In: International Symposium on Intelligent Data Analysis. pp. 97–109. Springer (2026)
15. Kiossou, H., Schaus, P., Nijssen, S., Aglin, G.: Efficient lookahead decision trees. In: International Symposium on Intelligent Data Analysis. pp. 133–144. Springer (2024)

16. Kiossou, H., Schaus, P., Nijssen, S., Houndji, V.R.: Time constrained dl8. 5 using limited discrepancy search. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 443–459. Springer (2022)
17. Kohler, H., Akrour, R., Preux, P.: Breiman meets bellman: Non-greedy decision trees with mdps. In: Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2. pp. 1207–1218 (2025)
18. Laurent, H., Rivest, R.L.: Constructing optimal binary decision trees is np-complete. Information processing letters **5**(1), 15–17 (1976)
19. Lin, J., Zhong, C., Hu, D., Rudin, C., Seltzer, M.: Generalized and scalable optimal sparse decision trees. In: International conference on machine learning. pp. 6150–6160. PMLR (2020)
20. van der Linden, J.G., Vos, D., de Weerdt, M.M., Verwer, S., Demirović, E.: Optimal or greedy decision trees? revisiting their objectives, tuning, and performance. arXiv e-prints pp. arXiv–2409 (2024)
21. Mazumder, R., Meng, X., Wang, H.: 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)
22. Murthy, S.K., Salzberg, S.: Decision tree induction: How effective is the greedy heuristic? In: KDD. pp. 222–227 (1995)
23. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J.: Learning optimal decision trees with sat. In: International Joint Conference on Artificial Intelligence 2018. pp. 1362–1368. Association for the Advancement of Artificial Intelligence (AAAI) (2018)
24. Nijssen, S., Fromont, E.: Mining optimal decision trees from itemset lattices. In: KDD. pp. 530–539 (2007)
25. Nijssen, S., Fromont, E.: Optimal constraint-based decision tree induction from itemset lattices. Data Mining and Knowledge Discovery **21**(1), 9–51 (2010)
26. Quinlan, J.: C4.5: Programs for machine learning. Elsevier (2014)
27. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C.G., Schaus, P.: Learning optimal decision trees using constraint programming. Constraints **25**(3), 226–250 (2020)