Time constrained DL8.5 using Limited Discrepancy Search

Harold Kiossou[1] [0000-0001-6972-9885] (✉), Pierre Schaus[1] [0000-0002-3153-8941], Siegfried Nijssen[1] [0000-0003-2678-1266], and Vinasétan Ratheil Houndji[2] [0000-0002-5467-9448]

[1] ICTEAM, Université Catholique de Louvain, Belgium
{first.last}@uclouvain.be
[2] LRSIA, Institut de Formation et de Recherche en Informatique, Benin
vratheilhoundji@gmail.com

Abstract. Decision trees that minimize the error on the training set with a depth limit have been found to be generally superior to those found by more standard greedy algorithms. However, when the search space to be explored is too large, the depth-first search used by exact algorithms can get trapped in left most branches. Consequently, when the user stops the algorithm, the best tree found so far may be unbalanced and poorly minimize the error. Our work aims to improve the anytime behavior by introducing the limited discrepancy search ingredient in these algorithms. This allows to explore the search space by waves increasingly deviating from standard heuristics such as information gain. Our experimental results show that the anytime behavior of the state-of-the-art exact method DL8.5 is greatly improved.

Keywords: Optimal Decision Trees · Limited Discrepancy Search · Knowledge Discovery.

1 Introduction

Decision trees are among the most popular models in machine learning. In particular, their simplest form, the boolean decision trees are considered in this paper since every dataset can be binarized. Each node represents an attribute or feature of the dataset, and each branch represents the selection made for the boolean attribute. The classification of a new instance is obtained by following the path from the root to the leaf node that gives the predicted class to the instance. Decision trees have become increasingly popular since their introduction [21]. Their simplicity, interpretability, and the number of algorithms to induce decision trees make them a preferred method for many applications. Learning a decision tree that minimizes the error on a training set is NP-hard. This is why, since their introduction, greedy algorithms have been used mainly to induce decision trees from a training set [7, 20]. Despite the lack of optimality guarantees, these algorithms, choosing top-down recursively the feature to split based on a heuristic such as information gain [15], offer a good trade-off between accuracy and scalability.

Recent advances in hardware and mathematical optimization libraries have made it possible to reconsider exact approaches to induce decision trees [5, 19]. Beyond the theoretical and algorithmic aspects, this field is getting more interest, mainly motivated by the fact that minimizing the error of the tree on the training set also allows to reduce the error on unseen data [5]. Several approaches based on Mathematical Programming [2, 5, 8, 19], Constraint Programming [18], and SAT solvers [16] have been proposed. Solver-based approaches are flexible and require less expertise to develop, but dedicated algorithms such as DL8.5 [3] and Murtree [9] based on branch-and-bound and dynamic programming have achieved the best results so far. They used a depth-first search to explore the search space of decision trees branching on the feature decision variables at each node. The performances for finding a provable optimal tree are generally good when the depth limit of the decision tree to discover is not too high (typically 3 or 4). However, for larger depths, when the training set is large and has many features, there is little hope to find and prove the optimal tree. In such cases, the search can get trapped in the left parts of the search tree exploration without having enough time to reconsider decisions close to the root. Stopping the search before its completion can therefore result in unbalanced decision trees (leaning to the left), with an error that is even larger than the ones the user would obtain with a greedy algorithm.

This work aims to improve the anytime aspect of the exact algorithm to induce decision trees by incorporating a limited discrepancy search [11], a well-known technique in combinatorial optimization, to improve depth-first search when an efficient heuristic is available. The article focuses on adapting the strategy to DL8.5 [3] but this idea can be applied to similar algorithms such as MurTree [9] or the AND-OR search Constraint Programming approach [18].

The adapted algorithm is called LDS-DL8.5. In this setting, the depth-first search also takes the decisions guided by a standard heuristic (information gain [15]) but does not allow to deviate too much from it according to a budget (called the discrepancy limit) per branch. With a discrepancy limit of zero, the algorithm discovers the same decision tree as C4.5. Then by gradually increasing the budget and restarting the search, the approach is able to deviate from the greedy tree and even possibly discover an optimal tree and prove its optimality when no pruning occurred because of the discrepancy limit. We show experimentally that the advantage of this approach is that the user can set an optimization time budget and still obtain a tree that is generally of better quality than the one obtained with a pure greedy algorithm such as C4.5 and CART. The trees discovered with LDS-DL8.5 are also in general better than those with DL8.5 for hard settings where it is not possible to find and prove optimality in a reasonable amount of time. Our implementation is publicly available at https://github.com/haroldks/lds-dl85.

This paper is organized as follows. In the next section, we present the related works, mainly works on optimal decision trees. We then explain the technical background by briefly discussing some notions of frequent itemset mining and the functioning of DL8.5 and the Limited Discrepancy search. Finally, we provide

some experimental results that show the efficiency and interest of LDS-DL8.5 w.r.t. DL8.5 and the state-of-the-art greedy algorithms CART and C4.5.

2 Related Works

Decision trees are built in most cases using heuristic algorithms such as CART [7] and C4.5 [20]. While highly scalable, the constructed tree may not be the most accurate, in particular in the presence of constraints, such as on the depth of the trees. Optimal decision tree algorithms aim to address this issue by exhaustively exploring the search space at runtime cost. They have seen a resurgence in prominence in recent years as algorithms and technology have improved. Most popular methods use a mixed integer programming-based approach. Bertsimas and Dunn [5] in their work encoded the problem of finding optimal decision trees with respect to misclassification error by fixing a maximum depth in advance and creating a number of variables to represent the predicates for each node. Verwer and Zhang [19] later proposed BinOCT, which reduces the number of variables and constraints present in the model by taking advantage of the binarization of the data. Aghaei et al. [2] suggested a MIP-based approach based on maximum flow formulation and Benders decomposition to tighten the relaxation of binary decision trees. Boutilier et al. [8] introduced valid inequalities for learning optimal multivariate decision trees. Other approaches, such as the work of Verhaeghe et al. [18], induced optimal decision trees with constraint programming principles. They developed models with maximum depth and minimum support constraints while using a branch-and-bound strategy to prune the search space.

Another class of methods used SAT solvers to induce optimal decision trees. Narodytska et al., [16] modeled the decision tree as a propositional logic to construct the smallest tree in terms of the total number of nodes that perfectly describes the given dataset. At first, a tree is learned by using some heuristic method. The SAT-solver is then called multiple times to find each time a perfect tree with one less node.

Researchers also develop specialized algorithms for decision trees. In their work, Nijssen and Fromont [17] developed DL8, an algorithm inspired by ideas from the pattern mining literature that can support a wide range of constraints. Their approach allowed to evaluate the different branches of a node individually while saving the obtained subtrees using a new caching technique to reuse them later. In a later work, Aglin et al. [3] developed DL8.5, an improved version of DL8. The main contributions are the introduction of an upper bound that limits the tree error allowed for a child node as soon as an optimal subtree has been determined for one of its siblings and a lower-bound technique that allows the algorithm to store information on both optimal and pruned subtrees to provide a lower bound on the tree error. These improvements lead to a method that outperformed previous approaches by several margins when used with a depth constraint. Demirovic et al. [9] advanced the DL8.5 algorithm by adding support to limit the number of nodes in the tree, an efficient procedure to compute the tree of depth two, and a novel similarity-based lower bounding approach.

3 Technical Background

DL8.5 induces boolean decision trees by relying on itemset mining concepts. It starts with the transactional dataset, where each transaction is an itemset indicating the existence or absence of each feature. Formally, it is defined as a collection $\mathcal{D} = \{(t, I, c) \mid t \in \mathcal{T}, I \subseteq \mathcal{I}, c \in \mathcal{C}\}$, where $\mathcal{T}$ represents the transaction sets or row identifiers, $\mathcal{I}$ is the set of possible items, and $\mathcal{C}$ is the set of class labels; within $\mathcal{I}$ there are two items (one positive, the other negative) for each original Boolean feature, and each itemset I contains either a positive or a negative item for every feature. As an illustration, Table 1b shows the transactional database representation of the binary matrix of Table 1a. The tids are the identifiers of the transactions, which can also be row numbers. For each itemset I:

- the cover of an itemset is the set of transactions that contain this itemset: $cover(I) = \{(t, X, c) \mid (t, X, c) \in \mathcal{D}\ and\ I \subseteq X\}$
- the class-based support of an itemset is the number of examples in its cover with the given class c: $Sup(I, c) = |\{(t, X, c') \in cover(I)\ and\ c = c'\}|$.

Table 1: Example of database formats.

(a) Binary Matrix.

Features			class
A	B	C	
1	1	1	1
1	1	0	0
1	0	1	1
1	0	0	1

(b) Transactional database.

tid	items	class
1	abc	1
2	$ab\neg c$	0
3	$a\neg bc$	1
4	$a\neg b\neg c$	1

DL8.5 Algorithm

For the sake of completeness, we first explain the DL8.5 algorithm and then introduce the changes related to the limited discrepancy search. Algorithm 1 shows the pseudocode of DL8.5. This algorithm, in the general case, performs a recursive depth-first branch and bound search at each node (itemset) to select the feature at that node that will extend the itemset. The left and right subtrees are collected, each obtained with a recursive call with the exclusion and inclusion of the considered item. The base cases ending the recursion occur:

- when the maximum depth constraint is reached, $\mid I \mid = maxdepth$ in line 7.
- when the node (itemset) error is already 0 ($leaf_error(I) = 0$) in line 7. The leaf error here is the misclassification rate, defined as : $leaf_error(I) = \mid cover(I) \mid - \max_{c \in \mathcal{C}} \{Sup(I, c)\}$.

Algorithm 1: $DL8.5(maxdepth, minsup)$

```
1  struct BestTree {ub : float; tree : Tree; error : float}
2  cache ← Trie < Itemset, BestTree >
3  bestSolution ← DL8.5 − Recurse(∅, +∞)
4  return bestSolution.tree
5  Procedure DL8.5 − Recurse(I, ub)
6      leaf_error ← leaf_error(I)
7      if leaf_error = 0 or |I| = maxdepth or timeout is reached then
8          if leaf_error ≤ ub then
9              return BestTree(ub, make_leaf(I), leaf_error)
10         return BestTree(ub, NO_TREE, leaf_error)
11     solution ← cache.get(I)
12     if solution was found and ((solution.tree ≠ NO_TREE) or (ub ≤ solution.ub)) then
13         return solution
14     (τ, b, base_ub) ← (NO_TREE, +∞, ub)
15     for all attributes i sorted by heuristic do
16         if |cover(I ∪ {i})| ≥ minsup and |cover(I ∪ {¬i})| ≥ minsup then
17             sol1 ← DL8.5 − Recurse(I ∪ {¬i}, base_ub)
18             if sol1.tree = NO_TREE then continue
19             sol2 ← DL8.5 − Recurse(I ∪ {i}, base_ub − sol1.error)
20             if sol2.tree = NO_TREE then continue
21             feature_error ← sol1.error + sol2.error
22             τ ← make_tree(i, sol1.tree, sol2.tree)
23             b ← feature_error
24             base_ub ← b − 1
25             if feature_error = 0 then break
26     solution ← BestTree(ub, τ, b)
27     cache.store(I, solution)
28     return solution
```

– when the itemset support is below a user-defined threshold on line 16.

In addition, DL8.5 uses an upper bound specified as a parameter of the recursion procedure. This bound at the root node is initially set to $+\infty$, but is tightened each time the algorithm finds a better tree. The update is made at line 21 and the algorithm will continue the search with this new bound (lines 17 and 19). The upper bound ensures the pruning of the search space using the test in line 18. Here, exploring the second branch for the current attribute is useless if the quality of the first branch tree is worse than the authorized upper bound. The error of the left subtree is also used to tighten the maximum error allowed for the right subtree in line 19.

As several recursion paths can lead to the same itemset and thus the same cover, DL8.5 avoids useless recomputation by storing the itemset, its associated upper bound and tree, even if no solution was found. In doing so, DL8.5 will not continue the search for a stored itemset and bound if it encounters it again later, considering that for the given bound, a good enough tree could not be found.

Weakness of DL8.5

DL8.5 may require high execution times for large datasets with high depth. The user can specify a timeout to limit the computation time and still get the

best tree found during this computation time. Unfortunately, this decision tree can be very unbalanced when the search gets stuck in the deepest branches, as illustrated in Figure 1. In this example, the time limit occurs when exploring the subtree below $\neg a$. As a result, all the examples that fell in nodes a had no chance of further splitting, and the error in this node can be quite large. Although DL8.5 is a very powerful algorithm when the search can be terminated with a chance to optimally split all nodes at best, this is not the case when the search is interrupted before the end of the search. For each successor of a node, the search procedure is called on the left and right branches. The depth-first search can get stuck for a dataset with many features and high depth in the recursive calls of the line 17. When a timeout occurs at line 7, the search will return all remaining successors without any further exploration, even when line 19 is called for the right branch. If an efficient heuristic is available, an extension of the depth-first search with limited discrepancy allows it to avoid getting stuck in the depths by allowing a certain number of deviations to each node according to a budget.

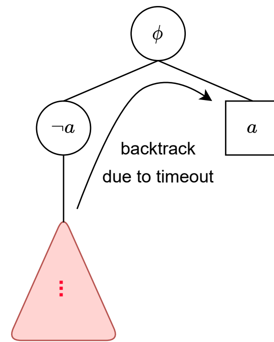


Fig. 1: Stuck DL8.5.

Limited Discrepancy Search

Many problem-solving approaches in AI use tree-depth-first search methods. It is common to employ a heuristic to guide the search towards the more promising search space regions first. For some problems, a good heuristic may directly lead to the optimal solution in the leftmost leaf node, but, in general, it has no guarantee of making no mistakes. This means that the search should have taken a few other rare decisions instead of always trusting the heuristic to discover the best solution.

By enumerating solutions in increasing order of the number of decisions that do not agree with the heuristic, the discrepancy search hopes to discover the best solution quickly. This strategy can be enforced using a depth-first search with a discrepancy budget along each branch, forcing the search to backtrack when

the limit is reached. The completeness of the approach is ensured by gradually increasing the discrepancy limit along the iterations. The first iteration does not allow any discrepancy and thus is only able to discover the leftmost leaf node without any backtrack. The next iteration allows one discrepancy and will enumerate all the possibilities with one allowed discrepancy, and so on for the subsequent larger discrepancy. Note that we can augment the limit by more than one increment between consecutive iterations to speed up the process, since each iteration may visit in theory a super-set[3] of the nodes at the previous iteration.

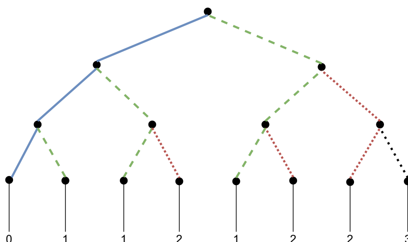


Fig. 2: Limited Discrepancy Search on a binary tree.

Figure 2 shows the result obtained with LDS searching for a binary tree of depth 3. The numbers at each leaf level give the total number of discrepancies needed to reach them. At the beginning of the search, the leftmost branch (blue) is traversed with a discrepancy of 0. The nodes of this branch correspond to the best results according to the heuristic used. The branches of the discrepancy 1 (green) are traversed when there is no solution at the discrepancy 0. At the root, it is possible to explore the right node with the discrepancy of 1 and then traverse the left branch for this node without a discrepancy budget as it corresponds to the best heuristic value.

Our main contribution is to include the LDS idea into the DL8.5 algorithm in the next section.

4 LDS-DL8.5

We propose LDS-DL8.5, a decision tree algorithm that improves the anytime behavior of DL8.5 by using limited discrepancy search. When there is not a time limit and the search is completed LDS-DL8.5 returns the optimal solution as the optimal approach like DL8.5. Algorithm 2 describes LDS-DL8.5. The main loop of the algorithm corresponds to the lines 5 to 9. There, the `Search` procedure is called with a discrepancy budget k that increases from 0 to a maximum value K determined by:

$$K = \sum_{i=0}^{d} \mathbf{A} - i - 1, \tag{1}$$

[3] The branch and bound may also prune the search space

Algorithm 2: LDS-DL8.5($maxdepth$, $minsup$, K)

```
1  struct NodeTree {ub : float; tree : Tree; error : float, discrepancy : int}
2  cache ← Trie < Itemset, NodeTree >
3  result ← NodeTree{+∞, NO_TREE, 0, 0}
4  k ← 0
5  while k ≤ K do
6      result ← Search(root, result.error, k)
7      if result.error = 0 or timeout is reached then
8          return result.tree
9      k ← augment_discrepancy(k, K)
10 return result.tree
11 Procedure Search(I, ub, k)
12     leaf_error ← leaf_error(I)
13     if leaf_error = 0 or |I| = maxdepth or timeout is reached then
14         if leaf_error ≤ ub then return NodeTree(ub, make_leaf(I), leaf_error, k)
15         return NodeTree(ub, NO_TREE, leaf_error, k)
16     node ← cache.get(I)
17     if node was found then
           /* The node is full explored */
18         if node.tree ≠ NO_TREE and node.discrepancy = K and ub ≤ node.ub then
19             return node
       /* List of the current node successors sorted by a heuristic */
20     successors ← get_successors(I, minsup)
       /* Node real discrepancy budget */
21     d ← discrepancy_limit(size(successors), maxdepth − |I|)
22     k ← min(d, k)
23     (τ, b, base_ub) ← (NO_TREE, +∞, ub)
24     for i in successors do
25         c ← successors.index(i)
26         if c > k then break // Discrepancy budget reached
27         first ← Search(I ∪ {¬i}, base_ub, k − c)
28         if first.tree = NO_TREE then continue
29         second ← Search(I ∪ {i}, base_ub − first.error, k − c)
30         if second.tree = NO_TREE then continue
31         feature_error ← first.error + second.error
32         τ ← make_tree(i, first.tree, second.tree)
33         b ← feature_error
34         base_ub ← feature_error − 1
35         if feature_error = 0 then break
       // Current discrepancy budget allows to reach the last successor or node is pure
36     if b = 0 or k = d then k ← K
37     result ← NodeTree(ub, τ, b, k)
38     cache.store(I, result)
39     return result
```

where $\mathbf{A}$ is the number of attributes in the dataset, and d is the maximum depth of the search. The algorithm can iteratively or exponentially increase the discrepancy with the function `augment_discrepancy`. By doing so, we are able to limit the number of times the algorithm can restart, which can improve the runtime. The search loop is stopped when the allowed execution time is reached or when a null-error optimal solution is found.

Increasing the discrepancies means increasing the number of successors that the algorithm is allowed to visit at each node. For the classical LDS, *zero discrepancies* consists in exploring at each level only the first attribute of the list returned by `get_successors`. The function returns the node successors based on

the minimum support constraints, sorted or not by a heuristic. If the time budget allows it and an optimal solution is not obtained, we progressively increase the discrepancy budget, allowing each node to explore more successors, each having a discrepancy value corresponding to its position in the list. When the discrepancy of an attribute exceeds the maximum allowed, the algorithm stops the search, as indicated on line 26. If the search can continue, the algorithm reduces the discrepancy budget allocated to this successor by removing its position from the current maximum. Moreover, contrary to the classical LDS, each branch of an attribute (for example, x and $\neg x$) has the same limit of discrepancy (lines 27 and 29) because an attribute is selected only if these two branches respect the imposed constraints.

The cost of a given iteration of LDS-DL8.5 is higher than the cost of the previous one due to the recomputing and re-exploration of previously visited nodes. To mitigate this cost, we use the cache by adding a parameter named *discrepancy* to the structure *NodeTree*. It corresponds to the discrepancy budget given to the node. This budget is re-evaluated on line 21. The actual number of successors along with the remaining depths ($maxdepth - |I|$) are used in the function `discrepancy_limit` to determine the current node discrepancy budget value using Equation 1. If the computed budget is larger than the passed budget, there is a high chance that this node will not be fully explored. The opposite means that the node will be fully explored. The real budget of the node is set to the minimum between the computed value and the allowed discrepancy budget for this node (line 22). When a node error is 0, or the computed budget is the same as the budget limit for that node not further exploration is needed its discrepancy is set to the maximum possible K in the *NodeTree* structure in line 36. The saved value avoids exploring the same nodes in the future iterations of the search if the upper-bound is worse than the stored value (line 18). Furthermore, a cached solution cannot be used unless it has not been proven to be optimal without a discrepancy limit. Thus, as long as the discrepancy is not set to the maximum value the node is explored.

With this scheme, the cost of each search iteration is reduced, mainly the latter one, because it is more likely that the left part of the search space will be fully explored from some value of the discrepancy.

5 Results

This section presents the results of various experiments that we conducted. Experiments were carried out to answer the following questions:

- **Q1** How does the performance of LDS-DL8.5 compare with DL8.5 and greedy algorithms in a time-limited configuration?
- **Q2** What happens when the DL8.5 recursion budget is limited to match a number of LDS-DL8.5 discrepancies?
- **Q3** How does LDS-DL8.5 perform compared to DL8.5 in the search for the optimal solution?

All experiments were carried out on 23 CP4IM datasets[4]. For the comparison of DL8.5 and LDS-DL8.5, the information gain was used as a heuristic. The algorithms were run on a server with an Intel Xeon Platinum 8160 CPU, 320 GB of memory, running Rocky Linux version 8.4.

To answer **Q1**, all algorithms were run with a minimum support of 1 and a maximum depth of 9. Each method runs with different time intervals of 30, 60, and 90 seconds on the whole dataset to compare the optimal methods with the greedy ones. Table 2 compares the error obtained by each algorithm according to the allowed time limit (TL). The datasets are sorted by their number of attributes (Feat.) and we show for each of them the number of transactions (Trans.) and the errors for each method. For LDS-DL8.5, two discrepancy augmentation schemes were used:

- `inc` where the discrepancy increases iteratively by one at each restart;
- `exp` where the discrepancy doubles at each restart.

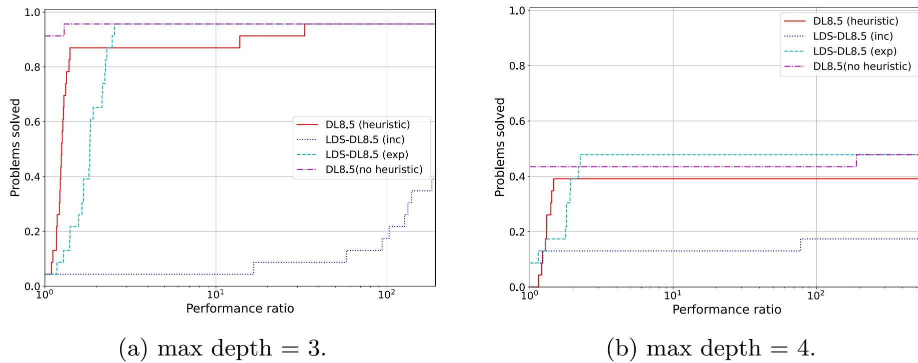
These tests show the efficiency of LDS-DL8.5 in a time-sensitive configuration. Regardless of the discrepancy augmentation scheme used, LDS-DL8.5 always has the lowest errors on the 23 instances. The greedy algorithms CART and C4.5 are fast enough to end in a few seconds. The errors will remain the same regardless of the time allocated to these algorithms. CART has the lowest performance among all algorithms, with a higher error in all instances except 2 where it was able to find the optimal solution (with an error of 0) together with the other methods, thanks to the heuristics used. C4.5 performs better than CART, as it was able to find the optimal solution for 8 instances. Next, DL8.5 and LDS-DL8.5 find the optimal solution on 12 instances, but DL8.5 has higher errors on the remaining 11 instances. Moreover, C4.5 performs better than DL8.5 on 10 instances. This confirms that DL8.5 might get stuck in the deeper branches of the left part of the search space, as it has to go through all the successors to select the best. This search costs time and, when the time limit is reached, DL8.5 will return the current node as the leave. On the contrary, LDS-DL8.5 ensures a minimal quality of the trees obtained. When run with a discrepancy limit of 0, the algorithm discovers the same tree as C4.5, allowing an immediate restriction of the upper bound for the next discrepancies and thus a better pruning of the search space. LDS-DL8.5(`inc`) generally has better results than LDS-DL8.5(`exp`) in this configuration. The way the discrepancy increases with `exp` allows the search to explore more nodes, increasing the risk of the search being stuck.

Regarding the question **Q2**, we have run LDS-DL8.5 with a limited number of discrepancies from 1 to 4. For each limit, the number of recursive calls is evaluated and defined as an additional constraint for DL8.5. The tests were carried out with the support of 1 and a maximum depth of 3. Table 3 summarizes the results of the experiment for the ten largest datasets in terms of features. The RB column corresponds to the recursion budget obtained by LDS-DL8.5 with the discrepancy limits mentioned in the *Disc.* column. Within the recursion

[4] https://dtai.cs.kuleuven.be/CP4IM/datasets/

budget, DL8.5 has greater difficulty in reducing the error, which remains higher than the one obtained by LDS-DL8.5. LDS-DL8.5 updates the error faster than DL8.5 and is more reliable for critical problems with time limits. Moreover, the trees obtained by LDS-DL8.5 are more balanced than those of DL8.5, as illustrated in Figure 4. This figure compares the trees obtained by LDS-DL8.5 for the discrepancy limits from 1 to 4 with those of DL8.5 with an equivalent recursion budget on the `mushroom` dataset. The trees obtained by DL8.5 are not balanced, unlike those of LDS-DL8.5. Furthermore, the trees do not change from discrepancy 1 to 3 with DL8.5 using the recursion budget, whereas LDS-DL8.5 will quickly improve the results. This is in line with the assumption that LDS-DL8.5 updates the upper bound and tree error more quickly.

To answer **Q3**, different algorithms were run to discover an optimal solution with a time limit of 10 min. Experiments were carried out with support of 1 and maximum depths of 3 and 4. For DL8.5, two versions were used: one with the information gain as heuristic and the second without any heuristic. The `inc` and `exp` versions of LDS-LD8.5 were also used. Figure 3 presents the performance profile [10] plots on the 23 instances with a maximum depth of 3 and 4 respectively. A performance profile is a cumulative distribution of the improved performance of an algorithm $s \in S$ compared to other algorithms of S over a set P of problems: $p_s(\tau) = \frac{1}{|P|} \times |\{p \in P : r_{p,s} \leq \tau\}|$ where the performance ratio is defined as $r_{p,s} = \frac{t_{p,s}}{\mathtt{min}\{t_{p,s} | s \in S\}}$ with $t_{p,s}$ the execution time of each algorithm.



(a) max depth = 3. (b) max depth = 4.

Fig. 3: Runtime performance profile plots.

In Figure 3a (depth = 3), D8.5 without heuristics has the best performance by solving the most problems in the least time. If a time factor of 2.5 is allowed, LDS-DL8.5(`exp`) solves the same amount of problem as the best solver. When the maximum depth is 4 DL8.5 without heuristics also has the best performance but in a time factor of 2.5 LDS-DL8.5(`exp`). LDS-DL8.5(`inc`) has the worst performance over time due to the higher number of restarts in this case. LDS-DL8.5(`exp`) is faster to prove optimality compared to LDS-DL8.5(`inc`) but will

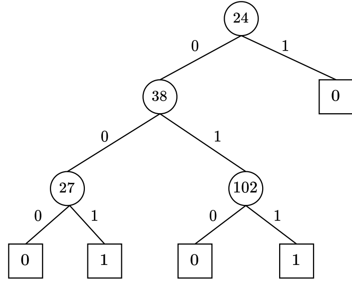
have more difficulties in the early time to update the error due to the large increase of the discrepancy budget at each iteration, leading to exploring more of the search space. This experiment shows that LDS-DL8.5 is able to find the optimal solution in a reasonable amount of time.

Table 2: Comparison of tree errors in the time-limited configuration for CART, C4.5, DL8.5 & LDS-DL8.5.

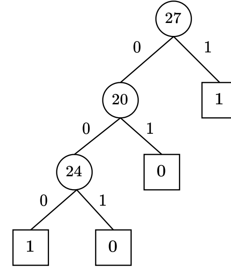
Datasets	Feat.	Trans.	TL(s)	Errors				
				CART	C4.5	DL8.5	LDS-DL8.5	
							inc	exp
ionosphere	445	351	30	26	**0**	**0**	**0**	**0**
			60	26	**0**	**0**	**0**	**0**
			90	26	**0**	**0**	**0**	**0**
splice-1	287	3190	30	258	21	68	**1**	**1**
			60	258	21	68	**1**	**1**
			90	258	21	68	**1**	**1**
vehicle	252	846	30	62	1	**0**	**0**	**0**
			60	62	1	**0**	**0**	**0**
			90	62	1	**0**	**0**	**0**
segment	235	2310	30	21	**0**	**0**	**0**	**0**
			60	21	**0**	**0**	**0**	**0**
			90	21	**0**	**0**	**0**	**0**
letter	224	20000	30	813	171	475	**37**	**37**
			60	813	171	475	**22**	37
			90	813	171	475	**22**	37
pendigits	216	7494	30	175	**0**	**0**	**0**	**0**
			60	175	**0**	**0**	**0**	**0**
			90	175	**0**	**0**	**0**	**0**
audiology	148	216	30	**0**	**0**	**0**	**0**	**0**
			60	**0**	**0**	**0**	**0**	**0**
			90	**0**	**0**	**0**	**0**	**0**
australian-credit	125	653	30	84	23	81	**3**	4
			60	84	23	81	2	**0**
			90	84	23	81	2	**0**
breast-wisconsin	120	683	30	24	1	**0**	**0**	**0**
			60	24	1	**0**	**0**	**0**
			90	24	1	**0**	**0**	**0**
mushroom	119	8124	30	544	**0**	**0**	**0**	**0**
			60	544	**0**	**0**	**0**	**0**
			90	544	**0**	**0**	**0**	**0**
german-credit	112	1000	30	265	120	174	29	**25**
			60	265	120	174	**22**	25
			90	265	120	174	**22**	25
diabetes	112	768	30	170	58	139	**18**	**18**
			60	170	58	49	**18**	**18**
			90	170	58	45	**18**	**18**
heart-cleveland	95	296	30	63	5	**0**	**0**	**0**
			60	63	5	**0**	**0**	**0**
			90	63	5	**0**	**0**	**0**
anneal	93	812	30	149	87	140	**68**	72
			60	149	87	140	**67**	72
			90	149	87	140	**67**	72
yeast	89	1484	30	436	251	432	184	**175**
			60	436	251	432	183	**173**
			90	436	251	432	175	**173**
hypothyroid	88	3247	30	54	34	63	**25**	**25**
			60	54	34	63	**24**	25
			90	54	34	63	**23**	25
kr-vs-kp	73	3196	30	189	18	54	**15**	**15**
			60	189	18	54	**15**	**15**
			90	189	18	54	**14**	15
lymph	68	148	30	18	**0**	**0**	**0**	**0**
			60	18	**0**	**0**	**0**	**0**
			90	18	**0**	**0**	**0**	**0**
hepatitis	68	137	30	15	**0**	**0**	**0**	**0**
			60	15	**0**	**0**	**0**	**0**
			90	15	**0**	**0**	**0**	**0**
soybean	50	630	30	50	12	31	**2**	**2**
			60	50	12	31	**2**	**2**
			90	50	12	31	**2**	**2**
vote	48	435	30	19	1	**0**	**0**	**0**
			60	19	1	**0**	**0**	**0**
			90	19	1	**0**	**0**	**0**
zoo-1	36	101	30	**0**	**0**	**0**	**0**	**0**
			60	**0**	**0**	**0**	**0**	**0**
			90	**0**	**0**	**0**	**0**	**0**
primary-tumor	31	336	30	40	24	59	**17**	**17**
			60	40	24	59	16	**15**
			90	40	24	59	16	**15**

Table 3: Experiments with recursion budget results on the 10 most large datasets in terms of features.

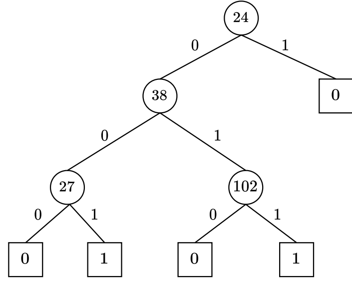
Datasets	**Disc.**	**RB**	**Errors**	
			DL8.5	**LDS-DL8.5**
ionosphere	1	50	**32**	**32**
	2	127	**32**	**32**
	3	265	**32**	**32**
	4	491	32	**30**
splice-1	1	63	574	**268**
	2	167	574	**268**
	3	358	574	**268**
	4	661	513	**267**
vehicle	1	55	216	**94**
	2	143	216	**76**
	3	261	216	**76**
	4	450	214	**63**
segment	1	24	**5**	**5**
	2	59	**5**	**5**
	3	121	**5**	**5**
	4	220	**5**	**5**
letter	1	53	**801**	813
	2	140	801	**686**
	3	236	801	**686**
	4	395	801	**686**
pendigits	1	57	88	**84**
	2	142	88	**60**
	3	268	88	**60**
	4	476	83	**51**
audiology	1	34	10	**6**
	2	87	10	**6**
	3	174	**6**	**6**
	4	319	6	**5**
australian-credit	1	51	**87**	**87**
	2	139	**87**	**87**
	3	304	**87**	**87**
	4	581	**87**	**87**
breast-wisconsin	1	61	50	**23**
	2	161	50	**16**
	3	297	30	**16**
	4	521	30	**16**
mushroom	1	37	376	**180**
	2	67	376	**180**
	3	128	376	**24**
	4	196	120	**8**



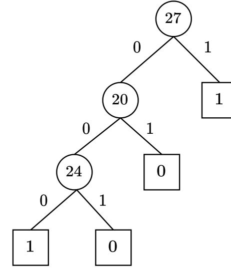
(a) LDS-DL8.5 tree with discrepancy limit 1.



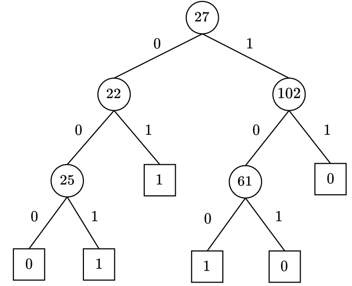
(b) DL8.5 equivalent tree of discrepancy limit of 2.



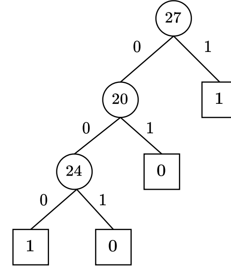
(c) LDS-DL8.5 tree with discrepancy limit 2.



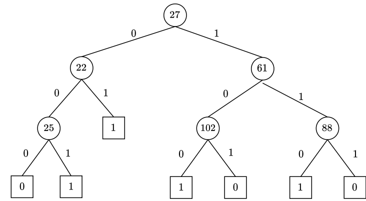
(d) DL8.5 equivalent tree of discrepancy limit of 2.



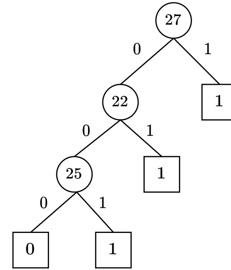
(e) LDS-DL8.5 tree with discrepancy limit 3.



(f) DL8.5 equivalent tree of discrepancy limit of 3.



(g) LDS-DL8.5 tree with discrepancy limit 4.



(h) DL8.5 equivalent tree of discrepancy limit of 4.

Fig. 4: Generated LDS trees with discrepancy limits of 1, 2, 3 and 4 with the DL8.5 equivalents on `mushroom` dataset.

6 Conclusion

This paper investigated the interest of using the limited discrepancy search to improve the anytime aspect of DL8.5. The LDS-DL8.5 algorithm, introduced in this paper, allows one to set low time limits and get good and balanced decision trees. Moreover, it mitigates the cost iteration by taking advantage of the cache, allowing the method to be sufficiently reliable when looking for optimal trees. Experimentation with 23 different datasets clearly showed the efficiency of LDS-DL8.5 w.r.t. DL8.5 and the state-of-the-art greedy algorithms CART and C4.5. LDS-DL8.5 is a reliable approach for finding good decision trees in a limited amount of time. As a future work, it could be interesting to study other restarting schemes such as the Luby strategy to improve LDS-DL8.5.

References

1. Aghaei, S., Azizi, M. & Vayanos, P. Learning Optimal and Fair Decision Trees for Non-Discriminative Decision-Making. *ArXiv:1903.10598 [cs, Stat]*. (2019)
2. Aghaei, S., Gomez, A. & Vayanos, P. Learning optimal classification trees: Strong max-flow formulations. *ArXiv Preprint ArXiv:2002.09142*. (2020)
3. Aglin, G., Nijssen, S. & Schaus, P. Learning Optimal Decision Trees Using Caching Branch-and-Bound Search. *Proceedings Of The AAAI Conference On Artificial Intelligence*. 3146-3153 (2020)
4. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. & Verkamo, A. Fast Discovery of Association Rules. *Advances In Knowledge Discovery And Data Mining*. pp. 307-328 (1996)
5. Bertsimas, D. & Dunn, J. Optimal Classification Trees. *Machine Learning*. **106**, 1039-1082 (2017)
6. Bessiere, C., Hebrard, E. & O'Sullivan, B. Minimising Decision Tree Size as Combinatorial Optimisation. *Principles And Practice Of Constraint Programming - CP 2009*. pp. 173-187 (2009)
7. Breiman, L., Friedman, J., Stone, C. & Olshen, R. Classification and Regression Trees. *Taylor & Francis* (1984)
8. Boutilier, J., Michini, C. & Zhou, Z. Shattering Inequalities for Learning Optimal Decision Trees. *International Conference On Integration Of Constraint Programming, Artificial Intelligence, And Operations Research*. pp. 74-90 (2022)
9. Demirović, E., Lukina, A., Hebrard, E., Chan, J., Bailey, J., Leckie, C., Ramamohanarao, K. & Stuckey, P. MurTree: Optimal Decision Trees via Dynamic Programming and Search. *Journal Of Machine Learning Research*. 1-47 (2022)
10. Dolan, E. & Moré, J. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*. 201-213 (2002)
11. Harvey, W. & Ginsberg, M. Limited Discrepancy Search. *Proceedings Of The 14th International Joint Conference On Artificial Intelligence - Volume 1*. pp. 607-613 (1995)
12. Hu, X., Rudin, C. & Seltzer, M. Optimal sparse decision trees. *Advances In Neural Information Processing Systems*. (2019)
13. Hu, H., Siala, M., Hebrard, E. & Huguet, M. Learning optimal decision trees with MaxSAT and its integration in AdaBoost. *IJCAI-PRICAI 2020, 29th International Joint Conference On Artificial Intelligence And The 17th Pacific Rim International Conference On Artificial Intelligence*. (2020)

14. Langley, P. Systematic and Nonsystematic Search Strategies. *Artificial Intelligence Planning Systems*. pp. 145-152 (1992)
15. Mitchell, T. Machine Learning. *McGraw-Hill Education*. (1997)
16. Narodytska, N., Ignatiev, A., Pereira, F. & Marques-Silva, J. Learning Optimal Decision Trees with SAT. *Proceedings Of The Twenty-Seventh International Joint Conference On Artificial Intelligence, IJCAI-18*. pp. 1362-1368 (2018)
17. Nijssen, S. & Fromont, E. Optimal Constraint-Based Decision Tree Induction from Itemset Lattices. *Data Mining And Knowledge Discovery*. pp. 9-51 (2010)
18. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C. & Schaus, P. Learning optimal decision trees using constraint programming. *Constraints*. pp. 226-250 (2020)
19. Verwer, S. & Zhang, Y. Learning Optimal Classification Trees Using a Binary Linear Program Formulation. *Proceedings Of The AAAI Conference On Artificial Intelligence*. **33**, 1625-1632 (2019)
20. Quinlan, J. C4.5: Programs for Machine Learning. *Morgan Kaufmann*. (1992)
21. Quinlan, J. Induction of decision trees. *Machine Learning*. 81-106 (1986)