# Learning Optimal Decision Trees Under Memory Constraints [⋆]

Gaël Aglin[0000−0002−6760−4752]✉, Siegfried Nijssen[0000−0003−2678−1266], and Pierre Schaus[0000−0002−3153−8941]

ICTEAM - UCLouvain, Louvain-la-Neuve, Belgium
firstname.lastname@uclouvain.be

**Abstract.** Existing algorithms for learning optimal decision trees can be put into two categories: algorithms based on the use of Mixed Integer Programming (MIP) solvers and algorithms based on dynamic programming (DP) on itemsets. While the algorithms based on DP are the fastest, their main disadvantage compared to MIP-based approaches is that the amount of memory these algorithms may require to find an optimal solution is not bounded. Consequently, for some datasets these algorithms can only be executed on machines with large amounts of memory. In this paper, we propose the first DP-based algorithm for learning optimal decision trees that operates under memory constraints. Core contributions of this work include: (1) strategies for freeing memory when too much memory is used by the algorithm; (2) an effective approach for recovering the optimal decision tree when parts of the memory are freed. Our experiments demonstrate a favorable trade-off between memory constraints and the run times of our algorithm.

**Keywords:** Decision trees · Optimization · Memory management.

## 1 Introduction

Decision trees (DTs) are among the most popular predictive machine learning models. They consist of tree structures in which internal nodes are labeled with tests and leaves are labeled with predictions. A key characteristic of DTs is their interpretability. A DT can be used to perform a prediction for an instance by performing a top-down traversal of the tree: the outcome of a test on an instance determines in which child node the traversal continues; the tests on the path towards a leaf explain why the model performs the prediction in that leaf.

Recent years have witnessed an increasing interest in explainable predictive models: in some crucial applications, such as medicine, bank loan repayment prediction or criminal recidivism prediction tools, it is important that the predictions of the models are understood in order to trust them.

One measure of the complexity of a DT model is its *depth*: the maximum number of tests performed to reach a leaf. Arguably, a DT model is easier to

[⋆] This work was supported by bpost

interpret if its depth is limited. Traditionally, such trees would be learned using heuristic algorithms such as CART. However, in 2017, Bertsimas et al. [5] showed that depth-limited *optimal decision trees* (ODTs) generalize better to unseen data than heuristic DTs. In this context, ODTs are decision trees that obtain the best possible accuracy on training data under a depth constraint.

Finding decision trees that optimize accuracy is not a trivial problem. For instance, [8] showed that the problem of identifying a DT under a size constraint is NP-complete. This has led to a renewed interest in developing efficient algorithms for finding optimal decision trees.

Approaches for finding ODTs can be put into two categories: (1) the category of approaches based on the use of mixed integer programming (MIP) solvers [13, 1]; (2) the category of specialized search algorithms that use some form of *dynamic programming* (DP) or *caching* [10, 11, 2, 12, 7, 9, 6]. Among these two categories, the specialized DP approaches have been demonstrated to be significantly faster [2, 6]. Unfortunately, however, an important drawback of the DP approaches is their high memory consumption: in the worst case their memory consumption is exponential in the number of features. MIP approaches, at least theoretically, do not have this weakness.

The key aim of this paper is to address this situation. We propose a new DP-based approach that allows a user to determine the trade-off between time and space, by introducing a parameter that determines the amount of memory the algorithm is allowed to use.

The intuition for the good run time performance of DP approaches is that these approaches do not perform a search for *trees*, but a search for *paths*. As the number of paths is smaller than the number of trees, this reduces the size of search space in DP algorithms significantly. Their high memory consumption derives from the fact that the DP-based algorithms store information about paths in a *cache*, from which they construct ODTs.

The contributions of this work are the following:

**C0** We study several caching strategies proposed in the literature, identifying which has the best characteristics when focusing on memory consumption.

**C1** We propose a simple modification of existing DP-based approaches that amounts to removing, from time to time, elements from the cache if the cache becomes too large.

However, removing elements from the cache can have two undesirable consequences: (1) an ODT can no longer be recovered from the cache, and (2) search can become slower, as we can no longer use results in the cache. Our next contributions address these critical problems.

**C2** We propose strategies for the order in which elements are deleted from the cache.

**C3** We present a strategy to recover deleted elements if these are required to output the ODT.

In the resulting approach, a parameter determines the trade-off between run time and memory consumption. Our final contribution (**C4**) is an experimental evaluation of this trade-off, as well as other dimensions important for the memory consumption of DP-based algorithms.

This paper is structured as follows. Section 2 presents the state of the art of different caching architectures used for ODTs learning and some search techniques used to reduce memory consumption. Then, Section 3 presents the technical details of caching systems implemented as a trie and a hash table. We present in Section 4 our new size-limited cache. Finally, present experimental results before concluding.

## 2 Related Work

There are two classes of approaches for finding ODTs: (1) approaches that rely on solvers, such as MIP solvers [13, 1], and (2) approaches that rely on dynamic programming [10, 11, 2, 12, 7, 9, 6]. The first class of methods relies on solvers of which the memory use is bounded; however, recent studies on DP-based approaches showed that the run time performance of these methods is significantly better. On the other hand, they suffer from a memory problem related to the use of a cache. Indeed, the size of the cache increases with the number of features and the depth of the ODT that needs to be found. For these algorithms it is hence important to minimize the amount of memory used. In the literature, two types of memory-based optimization have been studied. The first one focuses on the caching system itself and concerns the data structure used to implement the cache and the representation of the stored elements. The second optimization is related to the search and mainly consists in reducing the number of cache entries that are stored. Below we provide a high-level perspective on these optimizations; their details will be discussed in the next section.

### 2.1 Caching optimizations

In principle to identify an ODT, some form of exhaustive search needs to be performed. A core idea underlying the DP-based approaches is to store intermediate results during this exhaustive search. Using a cache key, these intermediate results are then used later on to avoid repeating the same search twice.

A key distinguishing factor between various approaches is the key that is used to associate intermediate results in the cache to.

The first algorithm to take this approach was DL8 [10, 11]. In DL8, a cache is built in which *itemsets* serve as the key to which information is associated. Here, every path in a decision tree corresponds to an itemset: an itemset is essentially the set of conditions on a path. For trees of limited depths, these keys are short. This technique is also used by the authors of an optimized version of DL8, DL8.5 [2], and works in the presence of depth constraints.

An optimisation was presented in DL8, in which *closed itemsets* were used. A closed itemset in this context is obtained by adding to an itemset $I$ all other conditions that hold in the same instances in which the conditions of $I$ are true. It was shown that this leads to more cache reuse; however, calculating this key takes more time and the key cannot be used in the presence of depth constraints, and hence was not used in DL8.5.

Keys based on instances were used in the GOSDT [9] and Murtree algorithms [6]. In particular, MurTree proposes to use as key a combination of instance identifiers and path length (of which more details will be provided in the next section). In particular for large datasets, this leads to much larger keys than a representation based on itemsets, but there are more opportunities for cache reuse.

Other differences between these algorithms concern the data structures used to store the cache and the keys within the cache. GOSDT and Murtree rely on the use of hash tables; GODST uses bit vectors to represent the sets of identifiers, while Murtree uses a list of IDs of instances.

DL8 and DL8.5, on the other hand, use a *trie*, aiming to exploit the overlap between itemsets that are stored in the cache.

None of these systems allow the user to impose a limit on the size of the cache.

### 2.2 Search optimization

The higher the number of elements to be stored, the larger the cache. Hence it is important to minimize the number of elements that need to be stored as much as possible. Various improvements have been studied to reduce the size of the search space and hence the number of elements in the cache.

One core is to use a form of branch-and-bound search to limit which parts of the search need to be considered. Aglin et al. [2] proposed a hierarchical upper bound and an infeasibility lower bound to avoid exploring some nodes of a search tree over paths. [7] proposes a similarity lower bound. In particular, a lower bound is derived for a path on the basis of the error found for another path. This is computed based on common and distinct instances of both paths. This bound is inspired by [4] and is also used in [6].

Using better bounds can effectively reduce the number of elements that need to be stored in the cache, and hence can have an impact on the memory used by the algorithm; however, while better bounds can make the search more feasible for larger datasets, they do not resolve the problem that for some datasets, the search algorithms will run out of memory.

An additional strategy for reducing the size of the cache was taken in the Murtree algorithm [6], which proposes a specialized algorithm for finding depth-2 ODTs without creating any cache entries. This reduces the number of elements needed to be stored to find an ODT.

The memory reduction technique proposed in this paper is not a search algorithm optimization. Rather, we propose an improvement to the cache system that allows to set an upper bound on the maximum number of cache entries. Hence, our optimizations are orthogonal to possible bounds used during the search, and we skip over most of the details of how bounds are used in these algorithms, even though also in our algorithm we use state-of-the-art bounds.

# 3 Caching in DP-based ODT learning

In this section, we describe how the trie and hash table are used to cache subproblem solutions when learning ODTs in existing algorithms; while doing so, we also contribute a comparison that will motivate our choices in this paper.

## 3.1 Caching in DP-based algorithms

Let $\mathcal{D} = \{\boldsymbol{x}, y\}^n$ be a binary dataset of $n$ instances and $\mathcal{F} = \{F_1, \dots, F_m\}$ be the set of $m$ features that describe $\mathcal{D}$. For each instance $\boldsymbol{x} = (x_1, \dots, x_m)$ in $\mathcal{D}$, $x_i$ takes value in $\{0, 1\}$. A decision tree recursively partitions a dataset into different groups following paths $p \in \mathcal{P}$. A decision tree can be seen as a collection $\mathcal{DT} \subseteq \mathcal{P}$ of paths, where each path starts at the root of the tree. While in decision tree, features are tested in a given order, it is the set of tests that determines which instances end up in a given node in the decision tree. For this reason, we will see a path as a *set* of tests on features. In the context of binary decision trees, a path $p$ is a set of tests over binary features, $p \subseteq \bigcup_{F \in \mathcal{F}} tests(F)$, where $tests(F)$ returns the two possible tests for the feature $F$, $F = 1$ (abbreviated with $f$) and $F = 0$ (abbreviated with $\overline{f}$); we assume $|p \cap tests(F)| \leq 1$ for all $F$.

At a high level, dynamic programming-based approaches for solving the ODT problem are based on recursive equations. For approaches that use itemsets as keys, this is the recursive equation in its simplest form:

$$min\_error(p) = \begin{cases} \min_{F \in \mathcal{F}} \sum_{t \in tests(F)} min\_error(p \cup \{t\}) & \text{if } |p| < maxdepth; \\ leaf\_error(p) & \text{if } |p| = maxdepth, \end{cases}$$

where the recursion starts at $min\_error(\emptyset)$. In other words, to determine the error made by a decision tree of minimal error, we need to pick the feature in the root of the tree that minimizes error, when summing up the lowest possible errors for the left-hand and right-hand subtrees. Note that in a tree the tests are ordered; in a tree we can first test $A = 1$ followed by $B = 1$, or, alternatively, first test $B = 1$ and then $A = 1$. This order is not important when determining $min\_error(\{A = 1, B = 1\})$. DP approaches are based on the idea of storing information for $p$, such as $min\_error(p)$, such that we can reuse the information for all possible orders in which the tests can be put in a tree.

For the leafs of the tree, we assume a prediction is made, and a class is associated. In the case of our paper, the associated class is the majority class, while the associated error is the misclassification error defined by $leaf\_error(p) = |\mathcal{D}| - max_{c \in \mathcal{C}} |\mathcal{D} : c|$, where $\mathcal{D}$ is the set of instances falling in the leaf $p$.

In its more complex form, the recursive equation can take into account other constraints, and can be rephrased to return the optimal tree itself as well.

DP-based algorithms perform a depth-first search using the recursive equation, reusing information that is stored already, and using bounds to limit the cases for which the recursion is executed.

For approaches that use instances as keys, the recursive equation is slightly different:

$$min\_error(\mathcal{D}, d) = \begin{cases} \min_{F \in \mathcal{F}} \sum_{t \in tests(F)} min\_error(\sigma_t(\mathcal{D}), d+1) & \text{if } d < maxdepth; \\ leaf\_error(\mathcal{D}) & \text{if } d = maxdepth, \end{cases}$$

where $\mathcal{D}$ is a dataset and $\sigma_t(\mathcal{D})$ selects the instances of $\mathcal{D}$ which satisfy the condition in test $t$. The recursion starts for the full dataset and depth $d = 0$. In other words, to determine the error of an optimal tree for a dataset $\mathcal{D}$, we need to determine which test to put in the root of the tree, such that error for the datasets resulting from the split is minimal. Compared to using paths as keys, instances can allow for more reuse, as multiple paths may select the same set of instances.

In this work our aim is to strictly monitor the memory consumption of DP-based algorithms. Hence it is important to understand which of these two approaches leads to better memory use; however, earlier studies did not address this question. We study this in the next subsection.

### 3.2 Comparison of Caching Strategies

Figure 1 shows a comparison of the memory consumption for different cache implementations for some datasets, when implemented in a state-of-the-art DP-based algorithm. The red curves represent the cache implementation that uses the set of instances as key. The other curves denote the cache implementation that uses paths as key. The difference between the green and blue curves will be explained in the next subsection. Notice that the number of cache entries for the

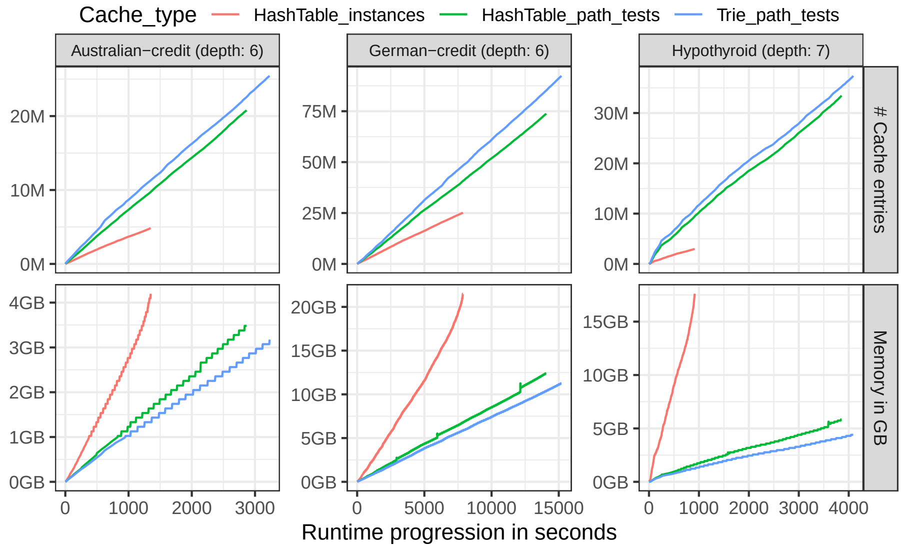

Fig. 1: Comparison of memory use given the cache type

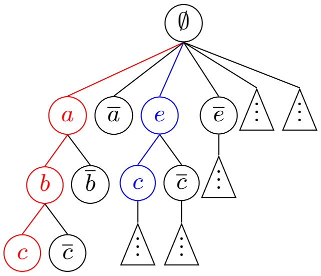

Fig. 2: Search space

Table 1: Hash table storage

| $\emptyset$ | | ⁂ | |
|---|---|---|---|
| $a$ | ⁂ | $\bar{a}$ | ⁂ |
| $e$ | ⁂ | $\bar{e}$ | ⁂ |
| $ab$ | ⁂ | $a\bar{b}$ | ⁂ |
| $ce$ | ⁂ | $\bar{c}e$ | ⁂ |
| $abc$ | ⁂ | $ab\bar{c}$ | ⁂ |

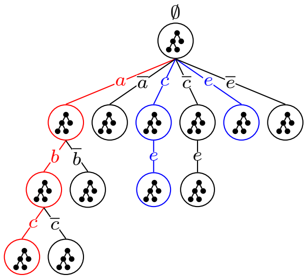

Fig. 3: Trie storage

instance-based representation is indeed low compared to the other approach. At the same time, it consumes much more memory as a key that consists of instance identifiers is much longer than a key that consists of tests. The rest of the paper focuses on the test-based representation, as it requires less memory than the instance-based one.

### 3.3 Caching data structure

To store the cache, DP-based algorithms require a data structure. Two implementations of the cache have been used in the literature. Most algorithms use a hash table, while DL8 and DL8.5 use a *trie* (or prefix tree).

The difference between these two data structures is illustrated in the following example, where a path is used as key; here we sort the tests in the path to obtain an ordered representation for the path. Let us assume a dataset with at least four (4) features and an ODT learning algorithm for trees of depth 3. Figure 2 shows a part of the search space to explore to find the ODT. Table 1 and Figure 3 show cache implementations based on a hash table and a trie, respectively. In the hash data structure, the path is passed through a hash function, but to avoid collisions, every path needs to be stored with its associated information. In the trie, nodes are created for every prefix of a path.

The size of these data structures can differ, as illustrated by the example. Consider the path $abc$ (in red) during the search; this path is reached in the search by first considering two other paths: $\{a, ab\}$. To save these three paths $\{a, ab, abc\}$ in the cache, in the case of a hash table, an entry is created for each path, and all tests in each path are saved. This is represented by red entries in Table 1: Six (6) tests must be saved to store the paths leading to $abc$. In the case of trie, the tests in common for parent paths are shared in such a way to avoid saving duplicate tests. To store $abc$ and its parent paths, only three (3) tests are necessary for a trie data structure, as the trie uses a compressed way to store paths. In practice, this is beneficial: the memory consumption in Figure 1 of the trie data structure (in blue) is lower than that of the hash table (in green), and motivates our choice for a trie in our experiments. However, note that in terms of the number of paths stored in the cache, the trie can be larger. For instance, this is the case for the paths leading to *ec* in Figure 2. While the hash table stores them using two cache entries, the trie needs three nodes for this operation. This explains why the number of cache entries in the trie is higher than in the hash table in Figure 1; however, as per entry in the hash table a complete path needs to be stored, memory consumption is higher; we prefer the representation with the lower memory consumption.

## 4 Learning ODTs with limited memory resources

In this section, we present our proposal to limit the memory consumption of DP-based learning algorithms.

Our core objective is to make sure the size of the cache is limited, while making sure that the performance of the search is not affected too much. Moreover, we wish to do so in a manner that can be integrated in DP-based algorithms with minimal modifications.

The core idea of our approach is simple: instead of keeping all elements in the cache, when necessary we will remove elements from the cache; when the search algorithm encounters these elements later, it will recalculate the results.

Hence, limiting the memory size will certainly impact the run time of the search algorithm, as its speed depends mainly on the reusability of the cache. The fewer entries in the cache, the less likely it is to find an existing solution. However, a good strategy specifying the order in which entries are deleted can limit the impact on the overall run time. In the next subsection, we will present our proposal for a bounded cache based on one deletion strategy. Subsequently, we will introduce additional deletion strategies and how to integrate them in the maintenance of the cache.

### 4.1 Implementation of a bounded cache

As the modifications of the DP-based systems we propose only concern the maintenance of the cache, we focus the description of our contribution to the maintenance of the cache. Pseudo-code for this can be found in Algorithm 1, for the trie data structure. Later, we explain how the pseudocode can be adapted to the simpler case of hash tables. Compared to the cache used by other systems, our new cache system requires two parameters. The first, `maxcachesize`, specifies the maximum number of entries that the cache can store. The second parameter; `wipeFactor` defines the percentage of the cache that will be cleared when the cache is full.

Any DP-based algorithm requires functionality that for a given key returns the data associated to the key, and if no such information is available, will add information to the cache; this is the function `insertOrGetEntry` in Algorithm 1. This function traverses the tests in a path, and either finds back the path in the trie, or adds the necessary nodes. Compared to the original trie-based system DL8.5, there are two differences in the insertion method. First, there is a condition (line 14) that checks that there is enough place in the cache to insert the

**Algorithm 1:** Class Bounded_Cache(maxcachesize, wipeFactor)

```
struct PathEntry{ useful: bool; curFeat: int; nDscPaths: int; solution:
    BestTree; childEntries: HashSet<int, PathEntry>}
struct BestTree {lb : float; feat : int; error : float }
// list of pairs <pathEntry, parentPathEntry>
deletionQueue $\leftarrow$ pair<PathEntry, PathEntry> [maxcachesize]
rootEntry, cachesize $\leftarrow$ newEntry(), 0
Method newEntry()
    return PathEntry(false,0, $-1$, (0, NO_FEAT, $+\infty$), {})
Method insertOrGetEntry(p : array of int) // p: path
    pEntry $\leftarrow$ rootEntry
    for t $\in$ p do // foreach test in path
        if t $\in$ pEntry.childEntries then
            pEntry $\leftarrow$ pEntry.childEntries.get(t)
        else // remainingTests() returns the number of tests in
            // the path which are not in the cache
            if cachesize + remainingTests() > maxcachesize then wipe()
            childEntry $\leftarrow$ newEntry()
            pEntry.childEntries.push({t, childEntry})
            deletionQueue.push({childEntry, pEntry})
            cachesize, pEntry $\leftarrow$ cachesize + 1, childEntry
    return pEntry
Method wipe()
    // mark nodes that required by the current state of the search
    setCurUsefulTag(rootEntry, {})
    countDscpaths(rootEntry)
    setOptiUsefulTag(rootEntry, {})
    sortDeletionQueue() // desc order with useful nodes at beginning
    nDel, counter $\leftarrow$ (int)(maxcachesize $*$ wipeFactor), 0
    for path, parPath $\in$ reverse(deletionQueue) do
        if counter = nDel or path.useful is true then break
        parPath.childEntries.remove(path); delete(path)
        counter, cachesize $\leftarrow$ counter + 1, cachesize $-$ 1
    unSetUsefulTag() // Remove tags
Method countDscpaths(pE : PathEntry)
    pE.nDscPaths $\leftarrow$ 0
    for t $\in$ pE.childEntries do
        pE.nDscPaths $\leftarrow$ pE.nDscPaths+countDscpaths(pE.childEntries.get(t))+1
    return pE.nDscPaths
Method getChildPathEntries(p : array of int, feat : int)
    $t_1, t_2$ $\leftarrow$ tests(feat)
    return {getEntry(sort(p $\cup$ $t_1$)), p $\cup$ $t_1$}, {getEntry(sort(p $\cup$ $t_2$)), p $\cup$ $t_2$}
Method setCurUsefulTag(pEntry : PathEntry, p : array of int)
    for pE, p $\in$ getChildPathEntries(p, pEntry.curFeat) do
        pE.useful $\leftarrow$ true; setCurUsefulTag(pE, p)
Method setOptiUsefulTag(pEntry : PathEntry, p : array of int)
    pEntry.useful $\leftarrow$ true
    for pE, p $\in$ getChildPathEntries(p, pEntry.solution.feat) do
        setOptiUsefulTag(pE, p)
```

path being created, otherwise the wipe function is called; this wipe function is responsible for removing elements from the cache. Second, each path added to the trie is also added to a deletion queue (line 17). This queue is maintained to keep track of the nodes in the cache, which is used when wiping the cache.

Since a node in a trie has a parent that links to it, when a path is deleted, it is necessary to delete the edge from the parent in the trie. For this reason, its parent path entry in the trie is stored to perform the edge deletion.

A critical part of our contribution is the wipe function. At its core, this function sorts the nodes in the cache, and subsequently deletes the desired number of nodes from the cache according to this order. Of critical importance is here how the nodes are sorted for removal. To determine this order, our wipe function first calls a number of functions to compute necessary information for the entries in the cache. This information is subsequently used when sorting the nodes. The remaining steps of the wipe function are straightforward. The deletion queue is traversed from the end to the beginning, and each path entry is deleted (line 26) until the number of entries to be deleted is reached, or all possible paths have been deleted (line 27). Note that the number of entries to delete is calculated according to the percentage provided by the wipeFactor parameter (line 25). Before deleting a path, the edge that connects it to its parent path is deleted (line 28) to inform the parent path that its child path no longer exists.

The information that is collected for elements in the cache is the following.

(1) Information concerning which paths the search is currently considering. Please remember that DP-based approaches perform a recursive search over paths, by adding tests to paths. In the process of calculating the result for the path $p$, they assume that path $p$ is present in the cache. The setCurUsefulTag method is used to mark the paths $p$ that are currently under evaluation. These paths will be put first in the order, and will never be deleted. Please note that the number of such nodes is very small. The setCurUsefulTag method uses a curFeat field in the entries in the cache. This field is initialized by a small modification of the recursive search function. For DL8.5 this is illustrated in Algorithm 2, where in green the code is indicated that is added in the recursive search function of DL8.5; parts of the code of DL8.5 that are not modified are skipped.

(2) Statistics concerning nodes in the cache. One such statistic which can be used to order elements, illustrated in our pseudo-code, is the number of descendants a node has in the trie. In the trie case, when we delete a node in a trie, we also need to delete its children in the trie. This implies that each path must be deleted before its parent path. As a parent has more descendants than its children, by ordering nodes on the number of descendants, we can assure children are deleted before their parents. Moreover, an intuition is that paths with numerous descendants are more expensive to evaluate than those with fewer descendant paths, and hence should be removed less quickly. To count the number of descendant paths per path, a simple post-order traversal is performed through the trie using the countDscpaths method. The result is stored in the variable nDscPaths (line 1).

(3) Information concerning which paths are part of the current optimal solution; we will return to this issue later on.

### 4.2 Different wipe strategies

In the pseudo-code above, the statistic we used to order nodes was *the number of descendant path entries in the cache*. We also consider the following alternatives.

*Number of reuses of solutions* The intuition behind the number of reuses of elements is that a path that has been reused many times is more likely to be needed again. This strategy removes the less frequently reused solutions before the more frequently used ones. To calculate the number of times a path solution has been reused, a variable initialized to 0 must be added to the path entry structure. Whenever in the method `insertOrGetEntry`, when an existing path is returned, the variable must be incremented for the path. Note that this deletion criterion does not satisfy the requirement of deleting the deepest nodes first in a trie. To enforce this behavior, there are two possibilities. The first is to set the method `sortDeletionQueue` so that the sort is performed according to two parameters. First, the length of the path and then the number of reuses. Another possibility is to increment the number of reuses for each ancestor of a path as well. In this work, we use the second option because it is based solely on the number of reuses, rather than using an additional criterion. Note that for this strategy, the statistic is not computed in the `wipe` method.

*All not required paths* This strategy, as the name implies, deletes all paths except from the *useful* ones, as defined by criterion (1) and (3) above. No variables need to be added to the path entry structure, and the deletion queue is not required. When the cache is full, after setting *useful* nodes, a post-order traversal is sufficient to delete all nodes without the *useful* tag.

### 4.3 Returning an ODT that relies on deleted elements

Until now, our recursive equations focused on returning the error of the most accurate tree. However, in practice we also wish to return the tree that obtains this error. In DL8 an approach was proposed that allows to do so with minimal additional memory use: for every path $p$, only the feature $F$ is stored in the cache that should be used to split optimally for $p$. The observation is that the optimal split for $p \cup \{f\}$ and $p \cup \{\overline{f}\}$ can also be found in the cache.

Unfortunately, if we wipe part of the cache, this strategy can no longer be used: if the optimal tree relies on a path that is no longer in the cache, we can no longer recover this tree from the cache.

One solution could be to associate a complete optimal tree to every path in the cache, but this would blow up the memory required for the cache, which we would like to avoid. Hence, a critical contribution of this work is an alternative solution that works well in practice: it avoids the use of large amounts of memory, while being fast at the same time. This solution consists of two components.

**Algorithm 2:** Bounded-DL8.5(maxdepth, maxcachesize, wipeFactor)

```
cache ← Bounded_Cache(maxcachesize, wipeFactor)
bestSolution ← DL8.5−Recurse({}, $+\infty$)
while tree(bestSolution) is incomplete do reconstituteWipedNodes({})
return tree(bestSolution)
// p is the path whose solution must be found
Procedure DL8.5−Recurse(p : array of int, ub : int)
    pEntry ← cache.insertOrGetEntry(sort(p))
    ... // if entry exists and has been solved, return its solution
    for each feature F in a well-chosen order and split in tests f and $\overline{f}$ do
        pEntry.curFeat ← F
        ... // compute error of best tree rooted by F
    pEntry.curFeat ← −1
    ... // return error and root of the tree with the lowest error
Procedure reconstituteWipedNodes(p : array of int, ub : int)
    pSol ← cache.insertOrGetEntry(sort(p)).solution
    if pSol.feat = NO_FEAT then return void
    ($pE_1, p_1$), ($pE_2, p_2$) ← cache.getChildPathEntries(p, pSol.feat)
    $\text{found}_{pE_1}$ ← $pE_1$ ≠ NULL and $pE_1$.solution.feat ≠ NO_FEAT
    $\text{found}_{pE_2}$ ← $pE_2$ ≠ NULL and $pE_2$.solution.feat ≠ NO_FEAT
    if $\text{found}_{pE_1}$ is false or $\text{found}_{pE_2}$ is false then
        if $\text{found}_{pE_1}$ is false and $\text{found}_{pE_2}$ is true then
            $pE_1$.solution.lb ← pSol.error − $pE_2$.solution.error
            DL8.5−Recurse($p_1$, $pE_1$.solution.lb + 1)
            reconstituteWipedNodes($p_2$)
        else if $\text{found}_{pE_2}$ is false and $\text{found}_{pE_1}$ is true then
            $pE_2$.solution.lb ← pSol.error − $pE_2$.solution.error
            DL8.5−Recurse($p_2$, $pE_2$.solution.lb + 1)
            reconstituteWipedNodes($p_1$)
        else
            $pE_1$.solution.lb ← 0
            DL8.5−Recurse($p_1$, pSol.error + 1)
            $pE_2$.solution.lb ← pSol.error − $pE_1$.solution.error
            DL8.5−Recurse($p_2$, $pE_2$.solution.error + 1)
    else
        reconstituteWipedNodes($p_1$)
        reconstituteWipedNodes($p_2$)
```

First, at the moment that we wipe the cache, using the `setOptiUsefulTag` function we determine which paths in the cache are part of the currently optimal solution; we do not remove these paths from the cache. In the best case, this optimal solution does not change any more and hence we can recover (most of) the solution from the cache.

Unfortunately it cannot be excluded that when the search continues, we find that a tree with a better quality exists that relies on paths that have been removed from the cache. In this case, we propose to *recalculate* the optimal tree for these paths. This algorithm is executed at the end of the original search to avoid making calculations for paths that later in the search may no longer be considered part of the final solution. This algorithm is described by the procedure `reconstituteWipedNodes` in the Algorithm 2. For each existing path, an attempt is made to obtain its two children paths from the cache (lines 16-18). If they exist (line 19), the DFS traversal continues (lines 34-35). Otherwise, the search is restarted. However, an important difference with the original search is that from the known errors (including the error of the optimal tree), much better bounds can be deduced than in the original search. In the case where a right child path exists but a search must be rerun for the left child path (lines 20-23), a simple subtraction between the errors of the parent path and the right child path provides the exact error of the left child path to be found. This error is used as a lower bound and a small value $\epsilon$ is added to it to define the upper bound. In the context of the misclassification rate, $\epsilon = 1$ is used. The same process is performed in the case of an existing left child path and a non-existing right child path (lines 24-27). When both child paths are nonexisting (lines 29-32), a search is performed for a first child path and the bounds for the second are derived from its solution. For the first path, the unknown lower bound is set to 0 while the upper bound is at most the error of the parent path added to $\epsilon$. This procedure to reconstruct the wiped paths of the final ODT is very efficient in practice.

### 4.4 Adapting to hash tables

The implementation of the bounded trie-based cache can easily be adapted to a hash table and is even simpler in this case. In order to sort and delete paths in a specific order, the deletion queue no longer needs to store pointers towards parents. Moreover, non-useful cache entries can be deleted in any order without hierarchy constraints, leading to a greater freedom in the choice of deletion criteria. For example, for the *reuses number* strategy that we propose, in the case of a hash table, we can rely only on this number to define the deletion order of paths without having to increment the *reuses number* of ancestor paths.

## 5 Results

In this section we answer five main questions:

**Q1** Which of the wipe strategies proposed has the lowest impact on the run time?
**Q2** What is the impact of the memory usage when using a bounded cache?
**Q3** What is the impact of a bounded cache on the run time?
**Q4** How fast is the algorithm to recover removed nodes from the ODT?
**Q5** How does the use of a bounded cache compare to a MIP approach?

The implementation of the learning algorithm without cache size restriction that we use in our experiments is `DL8.5` [2,3]. This means that the cache system used is a trie. However, we add some improvements to this `DL8.5` implementation to reduce the baseline memory consumption. These are the specialized algorithm proposed in MurTree to find depth-2 ODTs and the similarity lower bound introduced by OSDT. We call the final algorithm `DL8.5` in our experiments because the main search features originate from `DL8.5`. `Bounded-DL8.5` is the `DL8.5` version using our bounded cache. It is available at https://github.com/aia-uclouvain/pydl8.5. Experiments were run on a Linux Rocky 8.4 server with an Intel Xeon Platinum 8160 CPU @ 2.10Ghz and 320GB of memory.

In the first experiment, we use 20 binary datasets from CP4IM[1]. We run `DL8.5` on these datasets with a time limit of 5 minutes. To avoid comparing too easy instances, we learned ODTs of depth 5. Then we run `Bounded-DL8.5` with the same parameters until we find the ODT, or we reach the same point in the search as `DL8.5` when the timeout was raised. We show in Figure 4 two ratios of `Bounded-DL8.5` over `DL8.5`: the run time and the total number of entries (including those created after they have been removed). Each wipe strategy is considered for `Bounded-DL8.5`. For the wipe parameters, we consider 75% and 50% of the memory as value for the `maxcachesize` parameter and a wipe factor of 40% and of 30%. Note that the results shown are representative for the results for other choices of the parameters. It is interesting to notice that the time spent by the algorithm is proportional to the number of cache entries created. It can also be seen that the number of cache entries and the run time increase as the bound of the cache is restricted. On the other hand, it is difficult to observe a concrete trend in these values when a change is applied to the percentage of memory to free at each cache wipe. Note, however, that it becomes easy to answer **Q1** thanks to the Figure 4. As expected, the strategy of removing all non useful entries from the cache is the one increasing the most the run time of `Bounded-DL8.5`. Instead, the intuition of keeping as long as possible the entries often reused shows the best reduction of time impact. It performs better than removing the paths based on the number of descendants.

After this experiment, we select the best wipe strategy (*number of reuses*) to evaluate how `Bounded-DL8.5` can impact the memory usage on situations in which `DL8.5` requires a lot of memory to find the ODTs. To highlight these cases while ensuring reasonable run times, we experimentally select five specific datasets and depths. In the same way, we limited the cache size of `Bounded-DL8.5` to 30 million (30M) entries at most and set the percentage of entries to wipe to 40%. The impact of these values is already discussed above. The results are reported in Table 2. The memory usages are obtained by using the program *top* available on Unix operating systems. Notice that the memory needed to solve the problems using `Bounded-DL8.5` never reaches 10GB while up to 78GB is required using `DL8.5`. Regarding the dataset *anneal*, more than 600 million of entries are created with `DL8.5` to find an ODT of depth 8. The advantage of using `Bounded-DL8.5` is that this number can be reduced to a desired quan-

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

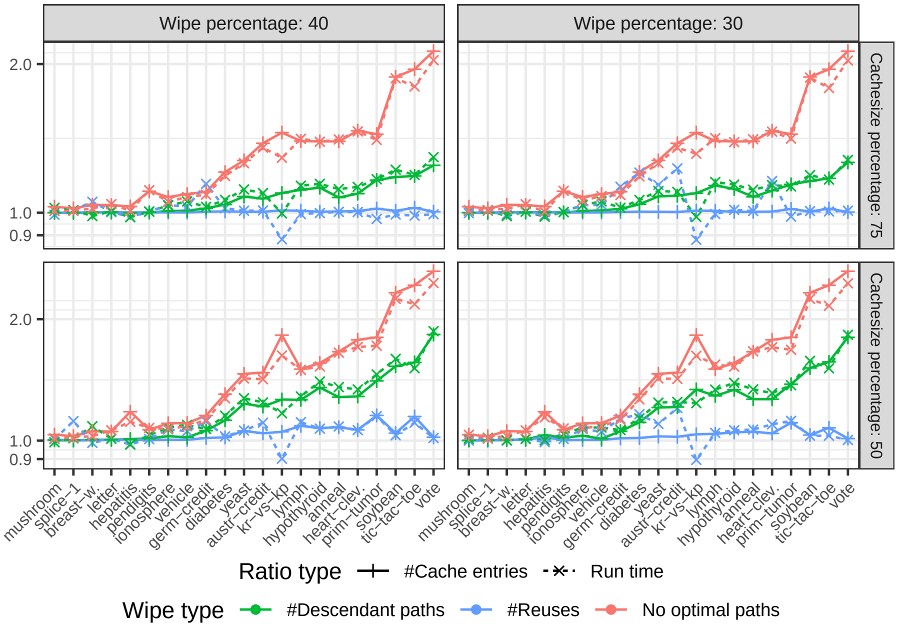

Fig. 4: Time factor of `Bounded-DL8.5` per wipe strategy

tity, here 30 million. As an answer to **Q2**, this reduces significantly the memory needed to find an ODT. Note, however, that this reduction depends on the limit on the number of cache entries. Moreover, it also has an impact on the run time. The time factor is recorded in Table 2. To answer **Q3**, notice that our strategy managed to achieve a time factor less than 1.5 on an instance that lasts over 3 hours with `DL8.5`. In the worst case, it almost reaches a time factor of 3 on an instance that originally required 78GB, which it reduces to 9GB. Regarding our algorithm to recover the nodes of the ODT that have been removed during the search, the time used by our algorithm is reported in milliseconds in the column *rTime*. Notice that for all instances, our algorithm uses less than 5 milliseconds to recover the removed nodes from the ODT. This answers the question **Q4**.

To answer **Q5**, we finally compare `Bounded-DL8.5` to a MIP approach. For this, we use `BinOCT`[2] model. The model is solved using CPLEX[3] 22.1.0. As MIP approaches are time consuming, we set a time limit to 70000 seconds, which is greater than the maximum time used by `Bounded-DL8.5` to solve an instance. Notice in Table 2 that `BinOCT` does not manage to solve any instance in the allocated time. This is represented by TO (timeout). Moreover, notice that the memory used by `BinOCT` is greater than `Bounded-DL8.5` for all instances.

[2] https://github.com/SiccoVerwer/binoct
[3] https://www.ibm.com/analytics/cplex-optimizer

Table 2: Comparison of unbounded and bounded caches

| Dataset | nFeats | nInsts | Depth | DL8.5 | | | Bounded−DL8.5 | | | | BinOCT | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | \|Cache\| | Time(s) | Mem (GB) | Time(s) | Time factor | Mem (GB) | rTime (ms) | Time | Mem (GB) |
| Anneal | 93 | 812 | 8 | 648M | 21907.28 | 78.5 | 65453.05 | 2.99 | **9** | 2.63 | TO | 9.8 |
| Diabetes | 112 | 768 | 7 | 238M | 18890.09 | 28 | 31534.13 | 1.67 | **6.3** | 0.21 | TO | 10.5 |
| German-credit | 112 | 1000 | 6 | 92M | 15198.32 | 11 | 21898.92 | 1.44 | **5.1** | 0.10 | TO | 10.1 |
| Kr-vs-Kp | 73 | 3196 | 8 | 136M | 3865.28 | 16.1 | 6365.04 | 1.65 | **5.3** | 4.68 | TO | 35.8 |
| Yeast | 89 | 1484 | 7 | 419M | 34977.98 | 50.5 | 62483.77 | 1.79 | **7.8** | 4.70 | TO | 17.5 |

## 6 Conclusions

In this paper, we address the problem of the huge memory consumption required by caching based algorithms for learning ODTs. We propose a technique that can be added to existing caches to wipe a desired number of entries from the cache. This leads to a significant reduction in memory consumption, with a smaller impact on run time. We propose strategies to reduce the impact of the wipe on the overall run time. Finally, we show that our approach finds ODTs more quickly than MIP approaches, while also consuming less memory.

## References

1. Aghaei, S., Gómez, A. & Vayanos, P. Strong optimal classification trees. *ArXiv Preprint ArXiv:2103.15965.* (2021)
2. Aglin, G., Nijssen, S. & Schaus, P. Learning optimal decision trees using caching branch-and-bound search. *Proceedings Of AAAI.* **34** pp. 3146-3153 (2020)
3. Aglin, G., Nijssen, S. & Schaus, P. PyDL8.5: a library for learning optimal decision trees. *Proceedings Of The 29th Conference On IJCAI.* pp. 5222-5224 (2021)
4. Angelino, E., Larus-Stone, N., Alabi, D., Seltzer, M. & Rudin, C. Learning certifiably optimal rule lists for categorical data. *ArXiv Preprint ArXiv:1704.01701.* (2017)
5. Bertsimas, D. & Dunn, J. Optimal classification trees. *Machine Learning.* **106**,(2017)
6. 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.* **23**, 1-47 (2022)
7. Hu, X., Rudin, C. & Seltzer, M. Optimal sparse decision trees. *Advances In Neural Information Processing Systems.* **32** (2019)
8. Laurent, H. & Rivest, R. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters.* **5**, 15-17 (1976)
9. Lin, J., Zhong, C., Hu, D., Rudin, C. & Seltzer, M. Generalized and scalable optimal sparse decision trees. *ICML.* pp. 6150-6160 (2020)
10. Nijssen, S. & Fromont, E. Mining optimal decision trees from itemset lattices. *KDD.* pp. 530-539 (2007)
11. Nijssen, S. & Fromont, E. Optimal constraint-based decision tree induction from itemset lattices. *Data Mining And Knowledge Discovery.* **21**, 9-51 (2010)
12. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C. & Schaus, P. Learning optimal decision trees using constraint programming. *Constraints.* **25**, 226-250 (2020)
13. Verwer, S. & Zhang, Y. Learning optimal classification trees using a binary linear program formulation. *Proceedings Of AAAI.* **33** pp. 1625-1632 (2019)