SAT-based Encodings for Optimal Decision Trees with Explicit Paths

Mikoláš Janota[1,2][0000−0003−3487−784X] and António Morgado[1][0000−0002−5295−1321]

[1] INESC-ID/IST, U. de Lisboa, Portugal
[2] Czech Technical University in Prague, Czech Republic

Abstract. Decision trees play an important role both in Machine Learning and Knowledge Representation. They are attractive due to their immediate interpretability. In the spirit of Occam's razor, and interpretability, it is desirable to calculate the smallest tree. This, however, has proven to be a challenging task and greedy approaches are typically used to learn trees in practice. Nevertheless, recent work showed that by the use of SAT solvers one may calculate the optimal size tree for real-world benchmarks. This paper proposes a novel SAT-based encoding that explicitly models paths in the tree, which enables us to control the tree's depth as well as size. At the level of individual SAT calls, we investigate splitting the search space into tree topologies. Our tool outperforms the existing implementation. But also, the experimental results show that minimizing the depth first and then minimizing the number of nodes enables solving a larger set of instances.

1 Introduction

Decision trees play an important role in machine learning either on their own [6] or in the context of ensembles [5]. Learning decision trees is especially attractive in the context of interpretable machine learning due to their simplicity. However, despite this simplicity, minimization of decision trees is well-known to be an NP-hard problem [16,10]. Yet, smaller trees are likely to generalize better.

To learn trees, suboptimal, greedy algorithms are used in practice. With the rise of powerful reasoning engines, recent research has tackled the problem by the use of SAT, CSP, or MILP solvers [38,37,1,27]. Indeed, the state-of-the-art technology shows that many (NP) hard problems are often successfully solved. Conversely, such applications drive the reasoning technology by providing interesting benchmarks.

This paper, follows this line of research and proposes a novel SAT-based encoding. This encoding enables finding a decision tree conforming to the given set of examples with a given depth and number of nodes. A minimal tree is found by iterative calls to a SAT solver while minimizing size and depth.

Focusing not only on size but also on depth of the tree brings about opportunities for further analysis. Intuitively, more shallow trees are less likely to

over-fit. Indeed, modern packages such as Scikit [30] enable imposing a threshold on the depth, which users have to set manually. Also, a shallow tree is more likely to be interpretable by a human because less memory is required to keep track of a single branch.

The problem at hand is of challenging complexity. In practice, we may need to deal with a high number of features and examples, which brings the search-space of possible trees into extreme dimensions. Looking for an optimal tree means not only finding such tree but also proving that no smaller tree exists.

The SAT technology has recently shown a lot of promise in tackling difficult combinatorial questions, e.g. Erdős' discrepancy [22] or the Boolean Pythagorean triples problem [13], among others. Inspired by these results we also investigate the splitting of search-space based on the topology of the decision tree. The paper has the following main contributions.

1. It proposes a novel SAT-based encoding for decision trees, along with a number of optimizations.
2. Compared to existing encoding, rather than representing nodes it represents *paths* of the tree. This enables natively controlling not only the tree's size but also the tree's depth.
3. It shows that minimizing depth first and then size enables tackling harder instances.
4. It shows that search-space splitting by topologies enables tackling harder instances.
5. The implemented tool outperforms existing work [27]

2 Preliminaries

Standard notions and notation for propositional logic are assumed [36]. A *literal* is a Boolean variable (x) or its negation (denoted $\neg x$); a *clause* is a disjunction of literals a *cube* is a conjunction of literals. A formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses. General Boolean formulas are also considered constructed by using the standard connectives conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), bi-implication ($\leftrightarrow$). State-of-the-art SAT solvers typically accept input in CNF. Non-CNF formulas are converted to CNF by standard equisatisfiable clausification methods [31].

Several constraints in the paper also rely on *cardinality constraints* [34]. These are also turned into CNF through standard means, the implementation avails of the cardinality encodings in the tool PySAT [26,18].

2.1 Training Data

Standard setting of supervised learning is assumed [35]. Following notation and concepts of [27] we expect features to be binary (with values 0, 1). Non-binary features can be reduced to binary by unary or binary encoding. Analogously, classes are also binary (positive, negative).

Examples are defined on a fixed set of features 1..F given as two sets, one containing the negative examples ($\mathcal{E}^-$) and second containing positive examples ($\mathcal{E}^+$). The examples are assumed consistent, i.e. $\mathcal{E}^- \cap \mathcal{E}^+ = \emptyset$. We write $\mathcal{E}$ for the whole set of examples, i.e. $\mathcal{E} = \mathcal{E}^- \cup \mathcal{E}^+$. Each example consists of feature-value pairs. We write $\sigma(q, f)$ for the value of a feature f in an example q. We assume that all the examples are complete, i.e. $\sigma(q)$ is total on 1..F.

3 SAT-based Optimization of Decision Trees

The objective is to develop a propositional formula whose models are decision trees congruent with the given set of samples. Such model then is found by a call to an off-the-shelf SAT solver. As customary, we take the approach of optimizing by solving a series of decision problems. This means finding a decision tree with a certain size and diminishing the size until no such tree exists. Alternatively, other type of search can be used, e.g., binary or progression.

This paper targets *two* optimization criteria: *size* and *depth*. Minimizing any combination of the two may be potentially be of interest. Section 5 discusses the exact type of search used in the implementation.

The structure of binary trees guarantees a number of well-known properties. Any tree with n nodes has $(n+1)/2$ leaves and $(n-1)/2$ internal nodes. Further, n is always odd and the number of leaves is equal to the number of paths going from the root to a leaf. Our encoding heavily exploits this property:

Rather than modeling nodes of a tree, we model the set of unique paths from the root to leaves.

The optimization algorithm has two levels. At the first level, search is being carried out on the tree's size and depth. At the second level, the decision problem of finding a tree with such depth and size is solved via a SAT solver. The SAT solver is used in a black-box fashion, i.e., the problem is encoded into its propositional form and any off-the-shelf SAT solver may be used to solve it.

In the remainder of this section we focus on the decision problem, which is invoked with a given number of paths P (controlling size) and maximum allowed number of steps in a path S (controlling depth).

The steps in a path are numbered in the following way. In the first step, each path is in the root. In the last step of a path, the path goes from an internal node to a leaf. This means that if we are looking for a tree with a particular depth and particular number of nodes we set S and P accordingly. If we are looking only for a tree with minimal number of nodes but with an arbitrary depth, the value of S is set to $\mathsf{P} - 1$, which corresponds to the number of internal nodes.

3.1 Path-based Encoding

The encoding we propose models each path from the root to a leaf separately while imposing relations between them that guarantee that the paths form a binary tree. Throughout the paper, we use the convention that for a node labeled

Variable	Semantics	Range
g_s^p	Path p at step s **g**oes right=1/left=0	$p \in 1..\mathsf{P}, s \in 1..\mathsf{S}$
t_s^p	Path p at step s is **t**erminated	$p \in 1..\mathsf{P}, s \in 1..\mathsf{S}+1$
e_s^p	Path p at step s is **e**qual to path $p-1$	$p \in 2..\mathsf{P}, s \in 1..\mathsf{S}+1$
$a_{s,f}^p$	Path p at step s is **a**ssigned feature f	$p \in 1..\mathsf{P}, s \in 1..\mathsf{S}, f \in 1..\mathsf{F}$
m_q^p	Path p **m**atches an example q	$p \in 1..\mathsf{P}, q \in \mathcal{E}$
$m_{f,v}^p$	Path p **m**atches on value v for feature f	$p \in 1..\mathsf{P}, f \in 1..\mathsf{F}, v \in \{0,1\}$
c^p	Path p is **c**lassified as positive	$p \in 1..\mathsf{P}$

Table 1. Variables used in the encoding

by a feature f, the left child corresponds to the value 0 of f and the right child corresponds to the value 1 of f.

To model the tree, introduce a matrix of variables, where each row represents a path and each column represents a step in the path. The first row (the first path) is a path that only goes to the left—it is the leftmost path in the tree. Analogously, the last row (the last path) is a path that only goes to the right—it is the rightmost path in the tree. In general, the paths are ordered in the way they would be obtained by running DFS that goes to the left first.

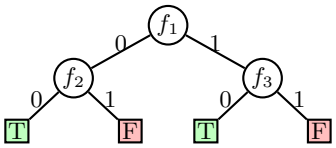
Each path corresponds to a sequence of 0's and 1's so that 0 is a step to the left and 1 is a step to the right. Then, we consider these paths in a lexicographic order. Each path is represented by a sequence of variables, one for each step, where the variable represents whether the path goes left or right in that step. Additionally, for each step we need to remember whether the path has already terminated and which prefix is shared with the previous path.

Table 1 summarizes the main variables of the encoding. The direction of each step s in a path p is determined by the variable g_s^p. What is somewhat unusual about this encoding is that paths may share prefixes. To that effect, the variable e_s^p represents that the path p in step s is in the same node as the preceding path $p-1$. The semantics of the variables e_s^p is defined inductively. All paths share the root and therefore e_1^p must be always true. In further steps, paths p and $p-1$ remain equal as long as both paths take steps in the same direction.

$$e_1^p, p \in 2..\mathsf{P} \tag{1}$$

$$e_{s+1}^p \leftrightarrow \left((g_s^p \leftrightarrow g_s^{p-1}) \wedge e_s^p \right), p \in 2..\mathsf{P}, s \in 1..\mathsf{S} \tag{2}$$

Since it is unknown beforehand how many steps are in either path, the variables t_s^p determine whether the path has already terminated or not. Observe that the variables t_s^p go up to step $\mathsf{S}+1$, whereas the variables g_s^p go only to step S. This is because the g_s^p variables correspond to edges in the path while termination is tracked for nodes (as well as equality). A terminated path remains terminated and cannot terminate if it is still equal to the previous one. Any path



(a) example tree

g_s^p	s	
p	1	2
1	0	0
2	0	1
3	1	0
4	1	1

(b) step direction

t_s^p	s		
p	1	2	3
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1

(c) termination

e_s^p	s		
p	1	2	3
2	1	1	0
3	1	0	0
4	1	1	0

(d) equality

Fig. 1. Assignment to the variables determining the tree's topology

must be terminated after the last step.

$$t_s^p \rightarrow t_{s+1}^p, p \in 1..\mathsf{P}, s \in 1..\mathsf{S} \tag{3}$$
$$t_s^p \rightarrow \neg e_s^p, p \in 2..\mathsf{P}, s \in 1..\mathsf{S}+1 \tag{4}$$
$$t_{\mathsf{S}+1}^p, p \in 1..\mathsf{P} \tag{5}$$

Example 1. Figure 1 shows a binary tree along with the values of the topology variables (g_s^p, t_s^p, and e_s^p). The tree is comprising 4 leaves, therefore 4 paths. In this simple example each path makes two steps and then it terminates. The second path shares everything with the first one except for the leaf. The third path only shares the root with the second path. The last path shares everything with the third path, except for the leaf. Observe that since this is a full binary tree, the g_s^p variables represent the binary numbers from 0 to 3.

Now it is necessary to ensure that the paths are lexicographically ordered. The first path always goes left and the last one always goes right. If a path p in step s is in the same node as path $p-1$, the path p can go left only if $p-1$ also went left (otherwise they would cross).

$$\neg g_s^1 \wedge g_s^{\mathsf{P}}, s \in 1..\mathsf{S} \tag{6}$$
$$e_s^p \rightarrow (g_s^{p-1} \rightarrow g_s^p), p \in 2..\mathsf{P}, s \in 1..\mathsf{S}+1 \tag{7}$$

The lexicographic order alone does not guarantee a correct topology. Since the tree is binary, any path must adhere to the following pattern. For a certain number of steps it shares the prefix with the preceding path until it breaks off. Once it breaks off, it has to go only to the left (or terminate). At the same time, the preceding path can only go right after the break-off point (or terminate). Otherwise, there would be a gap in the tree.

$$(\neg t_s^p \wedge \neg e_s^p) \rightarrow \neg g_s^p, p \in 2..\mathsf{P}, s \in 1..\mathsf{S} \tag{8}$$
$$(\neg t_s^p \wedge \neg e_s^p) \rightarrow g_s^{p-1}, p \in 2..\mathsf{P}, s \in 1..\mathsf{S} \tag{9}$$

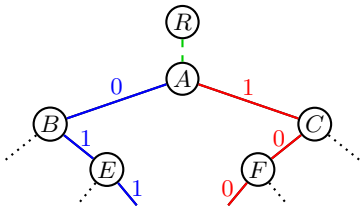


Fig. 2. Two consecutive paths R–E and R–F diverging in node A.

Figure 2 illustrates these constraints. Consider the blue path, $R \to A \to B \to E$ and the red path, $R \to A \to C \to F$, where the blue one is lexicographically smaller. The paths diverge in node A—blue goes left, the red goes right. Afterwards, the blue path may only go right or terminate. In contrast, the red path may only go left or terminate. The reason why this has to be the case is that for the red one to follow blue in our ordering, the blue one has to contain the *last* path for the subtree rooted in B while the red one has to contain the *first* path for the subtree rooted in C.

Assigning Features and their Semantics The encoding of semantics of the training data is similar to the one in [27] but with two major differences:

1. Here classification is only per *path*, while in [27] it is per *node* because any node can potentially be a leaf, which means semantics of the examples in our approach need only to be repeated $(n+2)/2$ times rather than n times.
2. Our encoding introduces explicit variables to track whether a given training example is matched for a given path, this is useful for one of the optimizations (see Section 3.2).

We make sure that each step is assigned exactly one feature and that no feature appears more than once on any path.

$$\sum_{f \in 1..\mathsf{F}} a^p_{s,f} = 1, p \in 1..\mathsf{P}, s \in 1..\mathsf{S} \tag{10}$$

$$\sum_{p \in 1..\mathsf{P}, s \in 1..\mathsf{S}} a^p_{s,f} \leq 1, f \in 1..\mathsf{F} \tag{11}$$

Recall that an example is seen as a set of feature-value pairs. We say that a feature-value pair f, v is *matched* on a path if the path makes a step in the direction of v in the node that is assigned the feature f. An example is matched if all its feature-value pairs are matched. These two concepts are modeled by the variables $m^p_{f,v}$ and m^p_q, respectively. Observe that f, v is also matched on any path that does not contain f at all. Finally, once a path matches any positive example, it must be classified as positive and the other way around.

$$m^p_{f,0} \leftrightarrow \bigwedge_{s \in 1..\mathsf{S}} (\neg t^p_s \wedge a^p_{s,f} \rightarrow \neg g^p_s) \qquad p \in 1..\mathsf{P}, f \in 1..\mathsf{F} \tag{12}$$

$$m^p_{f,1} \leftrightarrow \bigwedge_{s \in 1..\mathsf{S}} (\neg t^p_s \wedge a^p_{s,f} \rightarrow g^p_s) \qquad p \in 1..\mathsf{P}, f \in 1..\mathsf{F} \tag{13}$$

$$m^p_q \leftrightarrow \bigwedge_{f \in 1..\mathsf{F}} m^p_{f,\sigma(q,f)} \qquad p \in 1..\mathsf{P}, q \in \mathcal{E} \tag{14}$$

$$m^p_q \rightarrow c^p \qquad q \in \mathcal{E}^+, p \in 1..\mathsf{P} \tag{15}$$

$$m^p_q \rightarrow \neg c^p \qquad q \in \mathcal{E}^-, p \in 1..\mathsf{P} \tag{16}$$

Summary of the encoding. The constraints (1)–(16) are parameterized by natural numbers P and S and their satisfying assignments represent a sequence of P paths in a binary tree from the root to a leaf, where each path has at most S edges. The paths are lexicographically ordered, starting from the leftmost path and ending in the rightmost one. Additionally, the encoding ensures that there are no gaps between paths and therefore these represent the whole binary tree. Each node in a path is labeled by a feature in a way that shared prefixes among paths are labeled by the same features. Each path is assigned a classification class that must be congruent with the training examples given on the input.

3.2 Path encoding Optimizations

The encoding described above permits constructing any decision tree conforming to the given set of examples. However, certain optimizations can be made if we assume that we are not interested in superfluous nodes.

Enforcing example matching We make sure that any path (equivalently any leaf), *matches* at least one of the given examples. If it does not, its classification does not come from the examples and may therefore be arbitrary, which means it can be removed from any tree without violating the classification of the examples. At the formula level, additional constraints are added.

$$\bigvee_{q \in \mathcal{E}} m^p_q \quad p \in 1..\mathsf{P} \tag{17}$$

Pure features Features with the same value in all the examples can be ignored as they never permit distinguishing between two examples of a different class. This is done at the preprocessing level, so the encoder never sees them.

Quasi-pure features A feature may appear with a fixed value v within all the examples of one of the classes c. If such feature is assigned with the direction v, then the tree can immediately terminate with a leaf classified with c. As such, we

n	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
t	1	2	5	14	42	132	429	1,430	4,862	16,796	58,786	208,012	742,900	2,674,440	9,694,845

Table 2. Number of topologies (**t**) for tree size **n** ∈ 3..31 (Catalan numbers)

enforce that the child of a node assigned in the direction of the value v is a leaf classified with the class c. At the formula level, we add the following constraint for $s \in 1..\mathsf{S}$ and $p \in 1..\mathsf{P}$.

$$(a^p_{s,f} \wedge \mathrm{lit}(v, g^p_s)) \rightarrow (t^p_{s+1} \wedge \mathrm{lit}(c, c^p)) \text{ where } \mathrm{lit}(0, x) = \neg x, \mathrm{lit}(1, x) = x \quad (18)$$

Path lower bounds We propose to use MaxSAT to obtain lower bounds on the length of a path. The question we ask is what is the shortest possible path that separates positive and negatives examples. Since the lower bound considers only one path at a time, the order of features on that path is irrelevant. In preliminary experiments we have observed rather small lower bounds. However, the bound can be improved for the leftmost and rightmost branches. This gives us three types of bounds: for the leftmost and rightmost branches, and for any branch in between. In any path a feature either does not appear, or appears on a step that goes left or on a step that goes right. To model this behavior we introduce two variables for each feature x^0_f and x^1_f (similar to the dual rail encoding [25]). This corresponds to the following hard and soft constraints.

$$\begin{array}{lll}
\text{hard:} & \neg x^0_f \vee \neg x^1_f & f \in 1..\mathsf{F} \\
\text{hard:} & \neg x^1_f / \neg x^0_f & f \in 1..\mathsf{F}, \text{for leftmost/rightmost branch} \\
\text{hard:} & \bigvee_{f \in 1..\mathsf{F}} x^0_f \wedge \bigvee_{f \in 1..\mathsf{F}} x^1_f & \text{for general branch} \\
\text{hard:} & m^p_q \leftrightarrow \bigwedge_{f \in 1..\mathsf{F}} \neg x^{1-\sigma(q,f)} & p \in 1..\mathsf{P}, q \in \mathcal{E} \\
\text{hard:} & \bigwedge_{q \in \mathcal{E}^+} \neg m^p_q \vee \bigwedge_{q \in \mathcal{E}^-} \neg m^p_q & \\
\text{soft:} & \neg x^v_f & f \in 1..\mathsf{F}, v \in \{0, 1\}
\end{array}$$

4 Search-space Splitting by Topologies

Upon initial experiments, we observed that the SAT solver may struggle even on decision trees of modest size, e.g. 9 nodes. This is somewhat surprising because the number of topologies does not initially grow that much; see Table 2.

This suggests splitting the search space into individual topologies and call the SAT solver for each one of them separately. Like so, the SAT solver only needs to find the labeling of the tree. Intuitively this should be an easier problem because the SAT solver only needs to deal with one type of decisions.

This approach is not generally viable because eventually the number of topologies is too large. To which we propose the following approach. The upper part of the topology is fixed—until a certain depth—and the rest is left for the SAT solver to complete. This gives rise to *topology templates*. Each topology template is a tree, where each leaf is an actual leaf (□) of the topology or an incomplete subtree (△).

Algorithm 1: Topology enumeration, with □ - leaf, △ - subtree

```
1 Function TE (n, d) begin
2     if n = 1 then return {□}                                  // leaf
3     if d = 0 then return {△}                    // incomplete subtree
4     if d = 1 then
5         if n = 3 then return { tree (□, □) }
6         else if n = 5 then return { tree (△, □), tree (□, △) }
7         else return { tree (□, △), tree (△, □), tree (△, △) }
8     return {tree(l, r) | l ∈ TE(i, d − 1), r ∈ TE(n − i − 1, d − 1), i ∈ 1..n − 1}
```

Algorithm 1 recursively enumerates incomplete topologies on n nodes with the cut-off parameter d. In order to avoid repetitions in enumeration, certain cases need to be treated separately. If the cut-off parameter reaches 1, the children of the current node will either be leaves (□) or incomplete subtrees (△). This, in general gives three scenarios where either the left or the right child is a leaf and the second child is a subtree, or both are subtrees. However, in the case of $n = 3$, $n = 5$ the scenarios are different. Observe that because of the cut-off parameter, the generated topology template may have less than n nodes.

We study topology enumeration both for our encoding as well as the encoding of Narodytska et al. [27]. A given topology template in the encoding of Narodytska et al. is enforced by a cube corresponding to the child relation and the information whether a node is a leaf or not. An important property of our generation procedure is that the cut-off parameter is equal on all branches. This means that numbering the topology template by BFS gives the same numbers as a BFS on any topology corresponding to it. Since the encoding of `mindt` relies on BFS, this property lets us directly translate the relation into a cube.

Our path-based encoding does not allow easily encoding a topology template because the number of paths in an incomplete subtree (△) is unknown. To this

Algorithm 2: Topology enumeration with cardinalities

```
1 Function TE# (n, d) begin
2     if n = 1 then return {□}                                  // leaf
3     if d = 0 then return {#n}          // incomplete subtree of size n
4     return {tree(l, r) | l ∈ TE#(i, d − 1), r ∈ TE#(n − i − 1, d − 1), i ∈ 1..n − 1}
```

Algorithm 3: Measuring difference between topologies

```
1 Function TDiff (t1, t2, w) begin
2     if |t1| = 0 then return w|t2|
3     else if |t2| = 0 then return w|t1|
4     else return TDiff(t1.left, t2.left, wΔ) + TDiff(t1.right, t2.right, wΔ)
```

effect, we introduce a variation on the topology template where the leaves of the topology template are actual leaves ($\square$) or an incomplete subtree with a given cardinality ($\#k$). These topology templates can be easily enumerated as shown by Algorithm 2. Observe that the number of these topology templates may be larger than in the previous version. Such topology template is encoded into our path-based model in a straightforward fashion. Each path in the topology template fixes the direction in prefixes in a certain number of paths. The number of these paths corresponds to the $\#k$ node at the end of the path. Any path terminating in $\square$ corresponds exactly to one path in the path-based model.

4.1 Topology Enumeration

A cube describing a topology template can either be encoded into assumptions to enable *incremental SAT solving* [7] or appended as a set of unit clauses. We observed that in our case incremental solving does not pay off for hard instances. However, at the same time, if a large number of topology templates need to be examined, initializing a new SAT solver for each one of them is too costly. Therefore, the implementation employs both modes, incremental and non-incremental, depending on the number of topology templates to be examined.

Another point of interest is the order in which the topology templates are examined. In the case of non-incremental SAT solving and unsatisfiable instances, the order does not matter because all formulas need to be solved independently of one another. Hence, the order plays mainly a role in the case of satisfiable instances. The order heuristics we propose is the following.

We start with the assumption that we already have a suboptimal solution to the problem from a greedy (fast) algorithm. We would like to first focus on topologies that are similar to the topology of this suboptimal solution. In order to do so, we need some notion of *difference* between topologies (and topology templates). For this purpose we define a simple function that recursively compares the two topologies and accumulates a penalty once they are different. Additionally, subtrees with lower depth are accounted with less weight.

Algorithm 3 shows the function. If one of the given trees is empty, the penalty is the size of the other tree weighted by the factor w. Otherwise, the penalties are calculated as a sum of the left and the right subtrees, respectively. As the recursion descends, the weight is gradually decayed by the factor $\Delta \in (0, 1]$. In the implementation we chose the ad-hoc value of 0.75.

When partitioning the search space, the topology templates are enumerated in the increasing order of the difference from the suboptimal solution.

5 Experimental Evaluation

The tool was implemented on top of the PySAT package [18], which interfaces with a number of modern SAT solvers and provides a number of implementations of cardinality encodings. We used the *CaDiCaL solver* [3] and the *k-Cardinality Modulo Totalizer [26]*. This configuration was chosen after some careful preliminary experiments. We show that this configuration performs significantly better than the configuration used in the evaluation of Narodytska et al.

Our preliminary experiments also informed other ad-hoc choices that had to be made as the search and encodings can be configured in a large number of ways. An alternative would be to employ automated parameter tuning in the spirit of ParamILS [15]; we leave this as future work.

The SAT solver is used in a non-incremental fashion, i.e., every decision problem is solved independently of the other ones. The exception is topology enumeration: if the number of topologies is larger than 500, the incremental mode is employed (see Section 4).

A suboptimal greedy solution is obtained by the popular modern machine learning library Scikit-learn [30], which also enables a seamless integration with the Python implementation. The greedy solution is used in two scenarios: 1) to obtain an upper bound on the number of nodes in the solution 2) to inform the ordering of topologies during enumeration (see Section 4).

The experiments were performed on servers with Intel(R) Xeon(R) CPU at 2.60GHz, 24 cores, 64GB RAM, while always running 4 tasks in parallel. The time limit was set to 1000 seconds and the memory limit to 3 GB. The experimental results report on the following search modes:

(1) binary search on the number of nodes with no restriction on the depth without topology enumeration (with `sklearn` upper-bound)
(2) linearly increasing the number of nodes with no restriction on the depth with topology enumeration (linear UNSAT-SAT search)
(3) linearly increasing depth and linearly increasing number of nodes for each considered depth

Searches (1) and (2) find the smallest tree just as in [27]. The search (3) finds the smallest tree in the lexicographic ordering of the pair depth-size.

The evaluation was carried out on the benchmarks used in [27], kindly provided by Narodytska. These benchmarks were originally obtained by sampling a large set of instances [29], with sampling percentages of 20%, and 50% (we have used the same exact sampled benchmarks as Narodytska et al). The reader is referred to [27] for the details of the sampling procedure.

We compare our tools with the state-of-the-art tool `mindt` [27]. Our tool is run in the following configurations. The configuration `dtfinder` corresponds to the search (1), i.e. size minimization via binary search and path-based encoding. The configuration `dtfinder-DT1` is the same type of search but with encoding of Narodytska et al. The suffix `-T` in a configuration indicates topology-based

%	**0.2**				**0.5**			
nf./ns.	447/136				473/357			
#I	754				709			
	#nd	depth	cpu-time	#slv	#nd	depth	cpu-time	#slv
`mindt`	6	3	58	394	5	3	52	249
`dtfinder-DT1`	7	3	14	457	6	3	43	337
`dtfinder-DT1-T`	7	3	28	473	7	3	41	345
`dtfinder-DT1-T-O`	7	3	27	473	7	3	39	345
`dtfinder`	7	3	14	458	7	3	60	339
`dtfinder-T`	7	3	29	470	7	3	42	342
`dtfinder-T-O`	7	3	30	471	7	3	39	341
`d-dtfinder`	8	3	65	**519**	7	3	73	**352**
`d-dtfinder-T-O`	8	3	49	486	7	3	57	345
vbs	8	–	69	528	7	–	46	355

Table 3. Results on all the benchmarks divided by the percentage of random sampling.

search (search (2)). The suffix `-T-O` topology-based is search with heuristic ordering. The configuration `d-dtfinder` corresponds to the search (3), i.e. depth-size minimization.

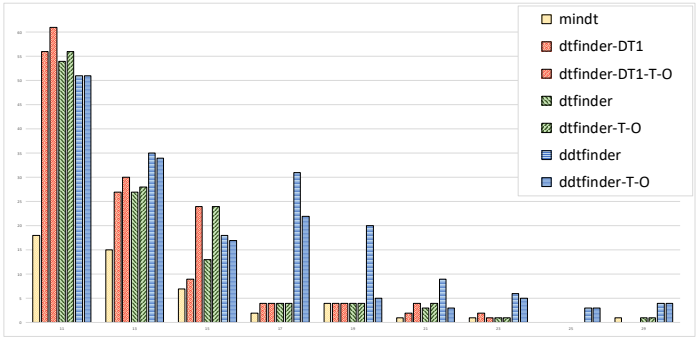
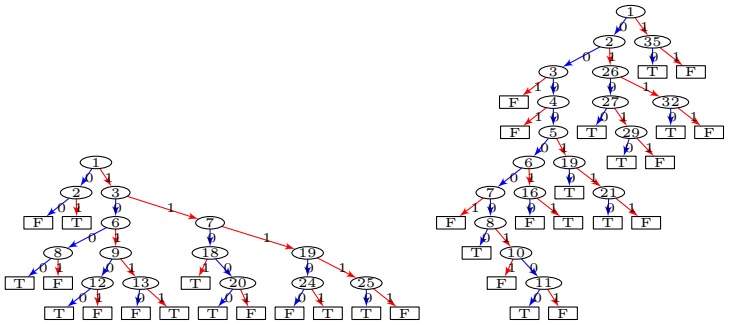


Fig. 3. Distribution of the sizes of calculated optimal trees

Table 3 summarizes results for all the considered benchmarks and tools. The first row (%) shows the percentage of random samplings used to construct the instance, the second row the average number of features (nf.) and samples (ns.). The third row (#I) shows the number of benchmarks in that category. The remaining rows are grouped according to the tool they represent.

For each of the tools we present four values: the average number of nodes discovered (#nd); the average depth of the tree reported (depth); the average



(a) `d-dtfinder` (b) `sklearn`

Fig. 4. `postoperative-patient-data-un_1-un` with 50% sampling

CPU time taken in solved instances (cpu-time); and the number of instances solved (#slv).

Figure 3 shows a histogram of the sizes of the optimal trees per solver. The vertical axis shows the number of solved instances and the horizontal groups the solvers according to the number of nodes of the reported decision trees. More detailed overview of the data can be found here on the authors' website [20].

Table 3 enables the following conclusions. Our implementation (`dtfinder`) outperforms the tool by Narodytska et al. (`mindt`) in all cases. This is also the case for their encoding; We attribute this to the choice of cardinality encoding and the SAT solver. We used k-Cardinality Modulo Totalizer and CaDiCaL while `mindt` uses sequential counter and glucose-0.3.

Comparing `dtfinder` with `d-dtfinder`, we can see that `d-dtfinder` is faster to compute a minimum depth solution than `dtfinder` is to compute a minimum size solution and even more interestingly, again solves even more instances.

The topology search-space splitting is beneficial in all encodings except for depth minimization. Both our encoding and encoding of Narodytska et al. solves more instances with topology enumeration. Not always this helps the average CPU time; however, it went from 60s to 39s in path-based encoding for the 0.5 instances. The ordering of topologies enables a minor speed-up but overall the effect is small.

The distribution of sizes of solved instances (Figure 3) shows that the hardness of an instance grows drastically with the size. While depth minimization is able to solve a handful of instances of size 29, the path-based encoding solves just 1, our implementation of Narodytska et al. none and, surprisingly `mindt` 1. This can be attributed to the number of topologies (see Table 2).

Overall, focusing on minimizing depth first is computationally advantageous, yet yielding decision trees of good quality. We illustrate this on a particular instance. Figure 4 shows decision trees calculated by our approach minimizing depth first (`d-dtfinder`) and calculated by the greedy approach (`sklearn`). The

optimal tree gives depth 6 and size 29, the greedy approach gives depth 11 and size 37. In contrast, the other approaches timeout on this instance in 1000 s.

6 Related Work

Greedy algorithms for learning decision trees based on recursive splitting are well-known [6,32,33]; see also [8] for an overview.

Various notions of optimality of decision trees appear in the literature. Some approaches focus on finding a tree with a *fixed depth* but with the best *accuracy* [38,37,1]. These approaches assume a full (perfectly balanced) binary tree of the fixed depth whose accuracy is to be optimized. While the problem is still very hard, it is in some sense easier because the topology is fixed and only the labeling needs to be calculated. However, combinations of these approaches in our approach is an interesting line of research.

Another approach is taken by [14], which optimizes a linear combination of accuracy and size. However, this approach is based on brute force search and in our experiments we were only able to synthesize trees with a handful of features while the considered benchmarks contain hundreds of features.

Closest to our work is [27], which uses SAT encoding to construct a size-optimal decision tree for a given set of consistent samples. In contrast to our work, individual nodes and their children relation are modeled explicitly. This means that a path from the root to a leaf is implicit. In principle, one could also restrict the depth of these implicit paths by adding additional counters or some other form of cardinality constraints. This is bound to be less efficient. Further, our encoding is closer to the idea of a tree. If the tree is modeled through nodes, it must be ensured that is in fact a tree via cardinality constraints—ensuring that each node has one and only one parent (except for the root) and that each internal node has two children. These cardinality constraints are not needed in our encoding. Since in our case, classes are per path rather than node we save half of the semantic constraint (see Section 3.1). It is interesting to compare how symmetries are broken in [27], where restrictions are imposed on the possible children nodes. In our approach paths are ordered lexicographically rather than in an arbitrary order. This order lets us single out the leftmost in the rightmost branches, which turned out to be useful in lower-bounding the depth (Section 3.2). We remark that lexicographic order is a popular means of breaking symmetries in general graphs, cf. [12].

Earlier work for minimization of decision tree using Constraint Programming (CP) exists [2]. It was shown in [27], that the approach by Narodytska et al. strictly outperforms the approach of Bessiere at al. this is most likely to be attributed to the fact that the CP encoding is asymptotically much larger.

Synthesis by calls to a SAT/SMT solver has seen increased interest in the recent years, cf. [21,19,28]. Haaswi et al. used topology enumeration to synthesize Boolean circuits [9]. The general idea is analogous to our approach (see Section 4). However, the set of possible topologies is partitioned differently. The possible topologies are DAGs, whereas they are trees in our case. Topologies in

their approach belong to the same partition if they have the same number of nodes at each level (levels are obtained by BFS). This approach is unlikely to give good partitioning for binary trees and is more expensive to encode than our approach. Further, in our approach, the enumeration of topologies simply goes over all possible topologies if the number of nodes is small.

The well-known technique of *cube-and-conquer* (CnC) splits the search-space by a lookahead solver [17,11]. The lookahead solver is run with a bound, which yields cubes to be decided by a traditional CDCL solver. Compared to our approach, CnC is much more general since it is applicable to any SAT instance, and, the lookahead solver is less likely to generate cubes that will be decided trivially. The downside is that CnC may not come up with a splitting as a human would. Further, the lookahead solver can be very costly. In our preliminary experiments, CnC performs much more poorly than a plain SAT solver on our instances. The order in which cubes are decided is also investigated by Heule et al. [11].

7 Conclusions and Future Work

This paper proposes a novel SAT-based encoding for decision trees, which enables natively controlling both the tree's size and depth. We also study search-space splitting by topology enumeration. Our implementation outperforms existing work of [27] but also enables a finer control due to the explicit representation of paths of the tree. This finer control lets us optimize practically interesting instances that had been out of reach.

The proposed approaches open a number of avenues for future research. The solving itself could be further improved by better splitting, parallelization, and combining with cube-and-conquer [11]. While some preprocessing of the examples was already used in our optimization techniques (Section 3.2), further inspection could be used to draw more information from them, e.g. introduction of extended variables in the spirit of [23]. The proposed techniques could also be integrated into more expressive approaches, e.g. SMT-based synthesis [21].

At the application level, we are investigating the integration of our tool with some greedy approaches, e.g. ensembles, where only limited depth is considered. Or, consider a hybrid between a greedy approach and an exact approach where an exact approach is invoked on smaller sub-problems. It would be interesting to investigate whether trees with a smaller depth are really easier to understand and interpret, and, what is the trade-off between depth and size. Our approach provides the means to exactly quantify these metrics.

The experimental evaluation shows that SAT solvers poorly handle a search-space with many topologies. We believe that this represents an important challenge for the SAT community.

Acknowledgements This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020, the project INFOCOS with reference PTDC/CCI-COM/32378/2017. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN with reference LL1902.

References

1. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning **106**(7), 1039–1082 (2017). https://doi.org/10.1007/s10994-017-5633-9
2. Bessiere, C., Hebrard, E., O'Sullivan, B.: Minimising decision tree size as combinatorial optimisation. In: Principles and Practice of Constraint Programming - CP. pp. 173–187 (2009). https://doi.org/10.1007/978-3-642-04244-7_16
3. Biere, A.: CaDiCaL, Lingeling, PLingeling, Treengeling and YalSAT entering the SAT competition 2017 (2017)
4. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
5. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
6. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth (1984)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003). https://doi.org/10.1016/S1571-0661(05)82542-3
8. Fürnkranz, J.: Decision Tree, pp. 330–335. Springer US, Boston, MA (2017). https://doi.org/10.1007/978-1-4899-7687-1_66
9. Haaswijk, W., Mishchenko, A., Soeken, M., Micheli, G.D.: SAT based exact synthesis using DAG topology families. In: Proceedings of the 55th Annual Design Automation Conference, DAC. pp. 53:1–53:6 (2018). https://doi.org/10.1145/3195970.3196111
10. Hancock, T.R., Jiang, T., Li, M., Tromp, J.: Lower bounds on learning decision lists and trees. Inf. Comput. **126**(2), 114–122 (1996). https://doi.org/10.1006/inco.1996.0040
11. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Eder, K., Lourenço, J., Shehory, O. (eds.) Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC, Revised Selected Papers. vol. 7261, pp. 50–65. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_8
12. Heule, M.J.H.: Optimal symmetry breaking for graph problems. Mathematics in Computer Science **13**(4), 533–548 (2019). https://doi.org/10.1007/s11786-019-00397-5
13. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the Boolean-Pythagorean triples problem via cube-and-conquer. In: Theory and Applications of Satisfiability Testing (SAT) (2016). https://doi.org/10.1007/978-3-319-40970-2_15
14. Hu, X., Rudin, C., Seltzer, M.: Optimal sparse decision trees. In: Neural Information Processing Systems NeurIPS (2019), http://papers.nips.cc/paper/8947-optimal-sparse-decision-trees
15. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009)
16. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is NP-complete. Inf. Process. Lett. **5**(1), 15–17 (1976). https://doi.org/10.1016/0020-0190(76)90095-8
17. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning SAT instances for distributed solving. In: Logic for Programming, Artificial Intelligence, and Reasoning

- 17th International Conference, LPAR-17. vol. 6397, pp. 372–386. Springer (2010). https://doi.org/10.1007/978-3-642-16242-8_27
18. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: A python toolkit for prototyping with SAT oracles. In: Theory and Applications of Satisfiability Testing - SAT. pp. 428–437 (2018). https://doi.org/10.1007/978-3-319-94144-8_26
19. Ignatiev, A., Pereira, F., Narodytska, N., Marques-Silva, J.: A SAT-based approach to learn explainable decision sets. In: International Joint Conference on Automated Reasoning (IJCAR) (2018). https://doi.org/10.1007/978-3-319-94205-6_41
20. Janota, M., Morgado, A.: (2020), http://sat.inesc-id.pt/%7Emikolas/dectrees
21. Kolb, S., Teso, S., Passerini, A., Raedt, L.D.: Learning SMT(LRA) constraints using SMT solvers. In: Lang [24]. https://doi.org/10.24963/ijcai.2018/323, http://www.ijcai.org/proceedings/2018/
22. Konev, B., Lisitsa, A.: Computer-aided proof of Erdős discrepancy properties. Artif. Intell. **224**, 103–118 (2015). https://doi.org/10.1016/j.artint.2015.03.004
23. Lagniez, J., Biere, A.: Factoring out assumptions to speed up MUS extraction. In: Theory and Applications of Satisfiability Testing - SAT. pp. 276–292 (2013). https://doi.org/10.1007/978-3-642-39071-5_21
24. Lang, J. (ed.): Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018. ijcai.org (2018), http://www.ijcai.org/proceedings/2018/
25. Manquinho, V.M., Flores, P.F., Silva, J.P.M., Oliveira, A.L.: Prime implicant computation using satisfiability algorithms. In: 9th International Conference on Tools with Artificial Intelligence ICTAI. pp. 232–239 (1997). https://doi.org/10.1109/TAI.1997.632261
26. Morgado, A., Ignatiev, A., Marques-Silva, J.: MSCG: Robust core-guided MaxSAT solving. JSAT **9**, 129–134 (2015)
27. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J.: Learning optimal decision trees with SAT. In: Lang [24], pp. 1362–1368. https://doi.org/10.24963/ijcai.2018/189, http://www.ijcai.org/proceedings/2018/
28. Narodytska, N., Shrotri, A.A., Meel, K.S., Ignatiev, A., Marques-Silva, J.: Assessing heuristic machine learning explanations with model counting. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT - 22nd. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_19
29. Olson, R.S., Cava, W.G.L., Orzechowski, P., Urbanowicz, R.J., Moore, J.H.: PMLB: a large benchmark suite for machine learning evaluation and comparison. BioData Mining **10**(1), 36:1–36:13 (2017). https://doi.org/10.1186/s13040-017-0154-4
30. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011), https://scikit-learn.org/
31. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)
32. Quinlan, J.R.: Induction of decision trees. Machine learning **1**(1), 81–106 (1986)
33. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
34. Roussel, O., Manquinho, V.M.: Pseudo-boolean and cardinality constraints. In: Biere et al. [4], pp. 695–733. https://doi.org/10.3233/978-1-58603-929-5-695
35. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Prentice Hall (2010)

36. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere et al. [4], pp. 131–153. https://doi.org/10.3233/978-1-58603-929-5-131
37. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C., Schaus, P.: Learning optimal decision trees using constraint programming. In: 31st Benelux Conference on Artificial Intelligence (BNAIC) (2019), http://ceur-ws.org/Vol-2491/abstract109.pdf
38. Verwer, S., Zhang, Y.: Learning optimal classification trees using a binary linear program formulation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, (AAAI). pp. 1625–1632. AAAI Press (2019). https://doi.org/10.1609/aaai.v33i01.33011624, https://www.aaai.org/Library/AAAI/aaai19contents.php