Learning Optimal Decision Trees using Constraint Programming

Hélène Verhaeghe[1], Siegfried Nijssen[1], Gilles Pesant[2], Claude-Guy Quimper[3], and Pierre Schaus[1]

[1]UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium, {firstname.lastname}@uclouvain.be
[2]Polytechnique Montréal, Montréal, Canada, gilles.pesant@polymtl.ca
[3]Université Laval, Québec, Canada, claude-guy.quimper@ift.ulaval.ca

Abstract. Decision trees are among the most popular classification models in machine learning. Traditionally, they are learned using greedy algorithms. However, such algorithms pose several disadvantages: it is difficult to limit the size of the decision trees while maintaining a good classification accuracy, and it is hard to impose additional constraints on the models that are learned. For these reasons, there has been a recent interest in exact and flexible algorithms for learning decision trees. In this paper, we introduce a new approach to learn decision trees using constraint programming. Compared to earlier approaches, we show that our approach obtains better performance, while still being sufficiently flexible to allow for the inclusion of constraints. Our approach builds on three key building blocks: (1) the use of AND/OR search, (2) the use of caching, (3) the use of the CoverSize global constraint proposed recently for the problem of itemset mining. This allows our constraint programming approach to deal in a much more efficient way with the decompositions in the learning problem.

Keywords: Decision Tree, CoverSize, AND/OR search tree

1 Introduction

Decision trees are popular classification models in machine learning. Benefits of decision trees include that they are relatively easy to interpret and that they provide good classification performance on many datasets.

Several methods have been proposed in the literature for learning decision trees. The greedy methods are the most popular ones [6, 15, 16]. These methods recursively partition a dataset into two subsets based on a greedily selected attribute until some stopping criterion is reach (such as a minimum number of examples in the leaf, or a unique class label in these examples). While in practice these methods obtain a good prediction accuracy for many types of data, unfortunately, they provide little guarantees. As a result, the trees learned using these methods may be unnecessarily complex, may be less accurate than possible, and it is hard to impose additional constraints on the trees, such as on the fairness of their predictions.

To address these weaknesses, researchers have studied the inference of *optimal* decision trees under constraints [1, 3, 4, 12–14, 19][1]. These approaches ensure that under well-defined constraints and optimization criteria, an optimal tree is found. Experiments conducted in earlier work [3, 13, 14] have shown that optimal decision trees computed with these exact methods can indeed obtain better classification performance while respecting constraints.

A problem that is solved by many of these earlier approaches [3, 13, 14, 19] is the following. Given a dataset in which all examples are binary; the problem is to find the decision tree that optimizes prediction accuracy, while enforcing a constraint on the depth of the decision tree.

The key ideas behind this constraint are that it limits the complexity of the decision tree, hence making the predictions of the tree easier to interpret while preventing over-fitting.

Several papers have studied the addition of other constraints to these approaches, including support constraints on the leafs of the tree [3, 14], on fairness [1], or on the preservation of privacy by these trees [14].

The main challenge that these methods need to address is that the problem of inducing decision trees under constraints is NP-hard [10]. Hence, approaches for this problem need to perform some form of exhaustive search through the space of possible trees. To explore this search space, earlier approaches have been built on existing technologies: Mixed Integer Programming (MIP) solvers, satisfiability (SAT) solvers, or itemset mining algorithms developed in the data mining literature.

This paper proposes a new, more scalable approach based on Constraint Programming (CP) for learning decision trees. Our approach combines these key ideas:

- the use of branch-and-bound in a CP solver to eliminate parts of the search space in which no solutions can be found;
- the use of the CoverSize global constraint, originally developed for itemset mining in CP, to calculate efficiently in which leafs examples end up [18];
- the use of an AND/OR search tree to exploit the fact that the optimal left-hand and right-hand subtrees of a node in a decision tree can be found independently from each other [8];
- the use of caching to store optimal decision trees for itemsets that have been considered in the past [13].

We will show that the combination of these different ideas leads to a model that is more efficient than other approaches proposed in the literature.

The paper is organized as follows. Section 2 presents the state of the art, followed by a formal definition of the problem in Section 3. Our CP model and CP search are detailed in Section 4. Finally Section 5 presents empirical results about our algorithm.

[1] The problem of embedding a decision tree as a constraint into a CP model has been studied in [5].

2 Related Work

Most related to this work are the alternative approaches for finding optimal decision trees. There is a number of alternative definitions for the problem of finding optimal decision trees, each using different constraints and optimization criteria.

The most popular setting studied in recent papers [1, 3, 19] is the one in which a decision tree of bounded depth is learned by maximizing the accuracy on a given training dataset. The limit on depth allows to model the problem as a MIP problem with a fixed number of variables. Constraints can be added, as long as they are linear; this includes constraints on fairness [1] or on the number of examples in the leafs [3]. We will use this problem setting in this work.

A slightly different setting was studied in the DL8 algorithm [14]. DL8 builds on top of itemset mining algorithms to find decision tree paths, and uses dynamic programming to build a decision tree from these paths. Effectively, it uses itemsets as the key of a caching data structure. As a consequence of the use of itemset mining, DL8 does not require a specific constraint on the depth of the decision tree; it uses a minimum support constraint to limit the size of the search space. This approach can be used on constraints that are not linear in nature. From this approach, we will adapt its link to itemset mining, and its use of caching.

To the best of our knowledge, CP has not yet been used in the setting where accuracy is optimized. Two earlier studies [4, 12] did however study the setting in which one finds the smallest decision tree consistent with a training dataset (i.e., the error of the decision tree has to be zero). As training data can be noisy and inconsistent, and hence finding a tree of zero error can be either impossible or undesirable, this setting is less common in the machine learning literature.

Similar to DL8, we will rely in this work on the fact that decision tree learning problems have many decompositions. We will exploit these using AND/OR search, which was studied extensively by Dechter et al. [8]. AND/OR search is not common in CP systems yet, and has not been used in decision tree learning yet; it has recently been exploited in the context of stochastic CP however [2].

3 Technical Background

3.1 Definition of the Problem

We restrict our attention to binary data. Continuous data can be discretized and binarized as proposed by Breiman et al. [6]; this observation was also exploited in earlier studies [13, 19].

We represent our data using an $n \times m$ binary matrix D. D_i represents the ith row of the data, or, following itemset mining terminology, the ith *transaction* of D. The number of transactions is thus n. The columns of the matrix represent the m *features* or *items* of the transactions. We assume in this work that each transaction belongs to one of two classes, represented by 0 and 1. Hence, the database can be split into D^+, a matrix of size $n^+ \times m$, containing all the

transactions from D associated to class 1, and D^-, a matrix of size $n^- \times m$, containing the ones associated to class 0.

In this work we are interested in finding decision trees. Each internal node w of a decision tree is associated to a feature (called the decision of the node) $d[w] \in \{1, \ldots, m\}$; each leaf is associated to a Boolean $b[w]$, representing the prediction for that leaf. We will use the function $F(r, t)$ to represent the predicted value for transaction t on a tree with root r, defined recursively as

$$F(w,t) = \begin{cases} b[w] & \text{if } w \text{ is a leaf;} \\ F(\text{left}(w), t) & \text{if } D_{t,d[w]} = 1; \\ F(\text{right}(w), t) & \text{if } D_{t,d[w]} = 0. \end{cases} \quad (1)$$

Here left(w) (resp. right(w)) returns the left-hand (resp. right-hand) subtree of node w.

We define the *depth* of a decision tree to be the maximum number of features on any path from the root of the tree towards a leaf. Given a maximum depth, our goal is to find a decision tree that minimizes the number of misclassified transactions (i.e., transactions where $v[t] \neq F(r, t)$):

$$\min \sum_{t=1}^{n} [F(r,t) \neq v[t]]. \quad (2)$$

We allow for the additional specification of a constraint on the minimum number of examples $N_{\min}$ in each leaf of the tree [3, 14],

An extension of the problem is to consider more than two classes (multi-class decision trees). We will limit our discussion to binary classes, but the extension towards data with more than two classes is relatively straightforward.

3.2 The CoverSize Constraint

To determine the accuracy of a decision tree, we need to decide in which nodes of the decision tree a transaction ends up. A correspondence can be drawn here with the *cover* of itemsets in itemset mining [13, 14]. We exploit this correspondence by adapting the CoverSize global constraint [18] to the context of learning decision trees. The original CoverSize has the following parameters: an array of Boolean variables (one variable for each feature), the database, and a counter variable, and is defined as follows:

$$\text{CoverSize}([I_1, \ldots, I_m], D, c) \Leftrightarrow c = |\cap_{I_i=1} \{t \in \{1, \ldots, n\} \mid D_{t,i} = 1\}|. \quad (3)$$

The goal of the constraint is to link an *itemset* to the number of transactions containing the itemset. The itemset is represented by the Boolean array $[I_1, \ldots, I_m]$: Boolean I_i is true if and only if feature i is included in the itemset. A transaction contains an itemset if and only if every feature in the itemset has value 1 in the transaction.

The dense representation of an itemset using a bit vector is unnecessary and impractical in our application. Instead, we will use a sparse representation:

$$\text{CoverSizeS}(\{K_1, ..., K_a\}, D, c) \Leftrightarrow c = |\cap_{i=1}^{a} \{t \in \{1, ..., n\} \mid D_{t,K_i} = 1\}| \quad (4)$$

This constraint has the following parameters: a set of integer variables $\{K_1, ..., K_a\}$ (each representing the identifier of a selected feature), the database and the cover counter. Similar propagation is possible for this constraint as for CoverSize.

Note that in the standard CoverSize constraint, we only test whether an item is included in a transaction ($D_{t,K_i} = 1$). In decision trees, we will also need to be able to test that an item is absent in a transaction. Neither with the initial CoverSize constraint nor its sparse version, is it possible to test for the absence of an item. To address this weakness, we propose the CoverSizeSR constraint, defined as follows:

$$\text{CoverSizeSR}(\underbrace{\{K_1, ..., K_a\}}_{\text{take set}}, \underbrace{\{L_1, ..., L_b\}}_{\text{drop set}}, D, c) \Leftrightarrow$$

$$c = \left|(\cap_{i=1}^{a}\{t \in \{1, ..., n\} \mid D_{t,K_i} = 1\}) \bigcap (\cap_{i=1}^{b}\{t \in \{1, ..., n\} \mid D_{t,L_i} = 0\})\right| \quad (5)$$

The *take* (resp. *drop*) set defines the features that should (resp. should not) appear in the counted transactions. This is also a straightforward modification of the original CoverSize constraint.

4 CP Modeling of the Problem

4.1 Model of the Problem

In this section, we will introduce the variables and constraints used in our model. Fig. 1 shows a visualization of our model for trees of a maximum depth of 3.

Note that in our model, we assume that a decision tree is a perfect tree. This assumption is motivated by the existence of a mapping of any proper binary tree (i.e., a tree where each node has exactly 0 or 2 children) into a perfect one (i.e., proper binary tree with all the leaves at the same level). We add a dummy feature f_0, not belonging to any of the transactions, to the model for unused decision nodes. A node with this value therefore has no transaction from the database on its left branch. Figure 2 shows how a proper tree can be made perfect by the use of the dummy feature.

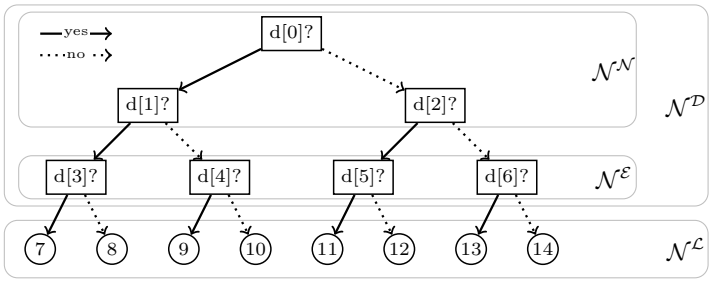
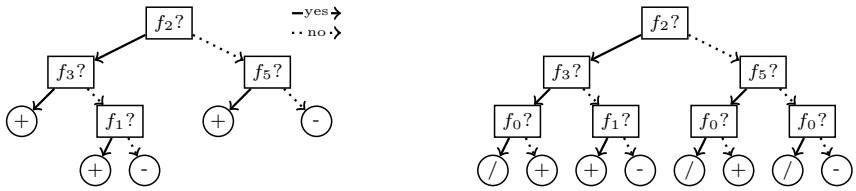


Fig. 1: Representation of a perfect decision tree of depth 3



(a) Proper binary tree (b) Equivalent perfect binary tree

Fig. 2: Example of the use of the dummy feature f_0 to transformed the proper binary tree into a perfect binary tree

The nodes ($\mathcal{N}$) of a perfect decision tree can be partitioned into two groups: the decision nodes ($\mathcal{N}^{\mathcal{D}}$), which are associated to a decision and which have children, and the leaves ($\mathcal{N}^{\mathcal{L}}$), which do not have children. The decision nodes ($\mathcal{N}^{\mathcal{D}}$) can be further partitioned into the end-nodes $\mathcal{N}^{\mathcal{E}}$, which do not have decision nodes as children, and the nodes $\mathcal{N}^{\mathcal{N}}$, which do. Variables and constraints are defined by the type of the node.

In our model, the number of variables and constraints are independent from the number of transactions in the database and the number of features. In fact, the number of variables and constraints only depends on the number of nodes in the tree.

Variables In our model we have variables with the following domains:

$$\text{dom}(d[i]) = \{0, 1, \ldots, m\} \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \tag{6}$$

$$\text{dom}(c^+[i]) = \{0, 1, \ldots, |D^+|\} \qquad \forall i \in \mathcal{N} \tag{7}$$

$$\text{dom}(c^-[i]) = \{0, 1, \ldots, |D^-|\} \qquad \forall i \in \mathcal{N} \tag{8}$$

$$\text{dom}(c[i]) = \{0\} \cup \{N_{\min}, N_{\min} + 1, \ldots, |D|\} \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \tag{9}$$

$$\text{dom}(e[i]) = \{0, 1, \ldots, \min\{|D^+|, |D^-|\}\} \qquad \forall i \in \mathcal{N} \tag{10}$$

Each decision node has a decision variable d (6) to model the decision feature. Its value can be 0 (representing the dummy feature f_0) or between 1 and m (representing one of the actual features f_1 to f_m). Two counters, c^+ (7) and c^- (8), are defined for each node of the tree. They are used to keep track of how many transactions respectively from D^+ and D^- match the decisions of the ancestors of the node. A third counter c (9), defined at the leaves, tracks the total number of transactions. The minimum number of transactions in each leaf is enforced by constraining the domain of c from $N_{\min}$ to $|D|$. Value 0 also belongs to the domain and is meant to be used only when the parent of the node is inactive (i.e. when its decision is f_0). An additional variable e (10), defined for each node, keeps track of the error of the sub-tree rooted at that node. Our model does not have an explicit variable for the class of the leaves. However, this can be easily deduced from the solution by taking the class associated with the highest counter.

Constraints On these variables, we define the following constraints:

$$c^+[i] + c^-[i] = c[i] \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (11)$$

$$c^+[i] = c^+[\text{left}(i)] + c^+[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (12)$$

$$c^-[i] = c^-[\text{left}(i)] + c^-[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (13)$$

$$e[i] = \min\{c^+[i], c^-[i]\} \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (14)$$

$$e[i] = e[\text{left}(i)] + e[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (15)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^+[i], D^+) \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (16)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^-[i], D^-) \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (17)$$

$$\textsc{AllDifferentExcept0}(\{d[j] \mid j \in \text{ancestors}(i)\} \cup \{d[i]\}) \qquad \forall i \in \mathcal{N}^{\mathcal{E}} \qquad (18)$$

$$d[i] \neq 0 \Rightarrow \min\{c^+[i], c^-[i]\} > e[i] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (19)$$

$$d[i] = 0 \Rightarrow (d[\text{left}(i)] = 0 \wedge d[\text{right}(i)] = 0) \qquad \forall i \in \mathcal{N}^{\mathcal{N}} \qquad (20)$$

First, constraint (11) links the counters at the leaves. Second, the counters at the decision nodes are linked to the counters of their children (12, 13). Third, the value of $e[i]$ is assigned to be the minimum between the class counters (14) at the leaves or to the sum of the errors from the children of i (15) for each of the decision nodes. To compute the values of the counters $c^+[i]$ and $c^-[i]$, we need to know which transactions match the decisions of the ancestors of the leaf. To this end, two CoverSizeSR global constraints (16, 17) are added at each leaf, one for each class. The decision variables of the ancestors (an ancestor is either the parent of a node, either the parent of an ancestor) are divided into two distinct sets: The *take* set $take(i) = \{d[j] \mid j \in \text{ancestors}(i) \wedge \text{left}(j) \in \text{ancestors}(i) \cup \{i\}\}$, containing the wanted features, and the *drop* set $drop(i) = \{d[j] \mid j \in \text{ancestors}(i) \wedge \text{right}(j) \in \text{ancestors}(i) \cup \{i\}\}$, containing the rejected features.

The next two constraints ensure the decision tree has no useless nodes. A node is useless if the decision taken in it was already taken in one of the ancestor nodes. An AllDifferentExcept0 (18) is used on the ancestors at each end-node to avoid this. A node is also useless if all the leaves below have the same class. This is avoidable if we constrain the error at the node to be strictly higher than the error of the subtree (19). Finally, when a decision node is inactive, all the decision nodes below should be inactive as well (20).

These constraints are enough to guarantee an optimal, well-formed tree (with no dummy decision feature being a parent from a non dummy decision and with no decision leading to only one classification).

Objective The objective is to minimize the sum of the errors at the leaves, which is stored in $e[\text{root}]$.

Redundant constraints We add a number of redundant constraints to make the search more efficient:

$$\text{dom}(c[i]) = \{0\} \cup \{N_{\min}, N_{\min}+1, \ldots, |D|\} \qquad \forall i \in \mathcal{N} \quad (21)$$

$$c^+[i] + c^-[i] = c[i] \qquad \forall i \in \mathcal{N} \quad (22)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^+[i], D^+) \qquad \forall i \in \mathcal{N} \backslash areRight(\mathcal{N}) \quad (23)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^-[i], D^-) \qquad \forall i \in \mathcal{N} \backslash areRight(\mathcal{N}) \quad (24)$$

$$c^+[i] < N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \quad (25)$$

$$c^-[i] < N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \quad (26)$$

$$c[i] < 2N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \quad (27)$$

$$d[i] \neq 0 \Rightarrow (c[\text{left}(i)] \geq N_{\min} \wedge c[\text{right}(i)] \geq N_{\min}) \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \quad (28)$$

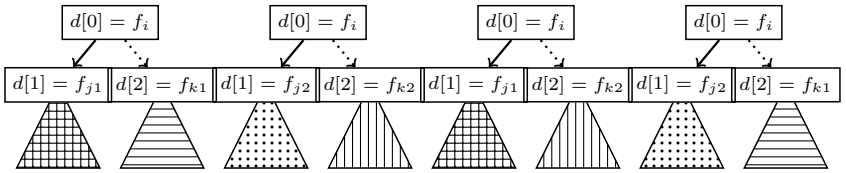
Here, $areRight(\mathcal{N}) = \{i \mid i \in \mathcal{N} \wedge i = \text{right}(\text{parent}(i))\}$; it represents the set of nodes being the right child of another node.

Adding a constraint CoverSizeSR for all of the nodes in the tree allows the computation of the exact values of the counters earlier in the tree and therefore helps prune earlier some candidate solutions. However adding them to all the decision nodes is not necessary. Constraints (12) and (13) can be relied on to compute the counters of one child based on the counters of the parent and the sibling. Constraints (23) and (24) are therefore used instead of (16) and (17). This allows a better propagation while using the same number of CoverSizeSR constraints. Constraints (25, 26) concern nodes with only transactions from one class left. When this arises, no decision should be taken in the node. As a minimum number of transactions should be in each activated node, if a given decision node does not have more than twice the threshold, no solution accepts a decision in the node (27). The contrapositives of (25), (26), (27) are also logically true. Combined together, they correspond to (28) which states that if the dummy decision is no longer in the domain, there should be enough transactions in each of the children. This constraint formulation requires to have the counter c (21) and the constraint linking the counters at each node (22).

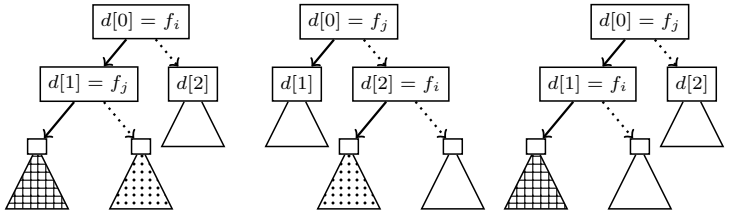
4.2 Search

The motivation behind the use of a specific search strategy is to exploit the tree-decomposition into subproblems. During search each node of the search tree is associated to a subtree of the decision tree being built. This subtree, identified by the node id `currProblem`, is always rooted on a decision node. The assignment of the decision variables occurs in top-down fashion. Therefore in a given node of the search tree, we can always assume every node in $ancestors($`currProblem`$)$ has been assigned. Algorithm 1 details the pseudo code of our algorithm.

Big picture. Our search is the composition of three techniques: AND/OR search trees, branch-and-bound optimization, and memorization. Each of them aims to answer one of the specificities of the problem.



(a) Independence of subtrees



(b) Redundant subtrees; identically highlighted subtrees are identical

Fig. 3: Decompositions

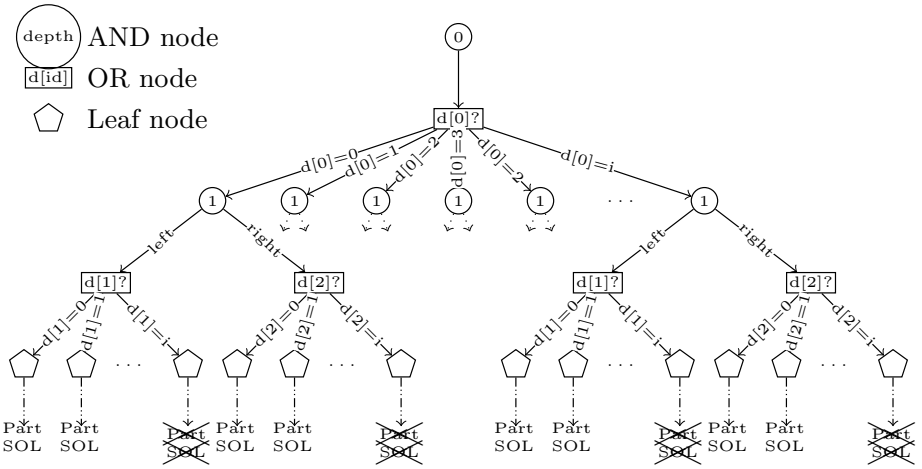


Fig. 4: AND/OR formulation of the search tree

Subtree independence. Given a subtree with its root decision and ancestors' decisions assigned, its two children are totally independent from one another. Any solution from the left child combined with any solution from the right child leads to a solution of the initial subtree. This is illustrated at Fig. 3a. However our goal is to find the best solution and not one solution. Moreover our objective function is the sum of a cost computed in each of the leaves, independently. Therefore, the optimal solution, given a root and ancestors' decisions already assigned, can be computed independently by computing the optimal left child, then the optimal right child and finally combine them. The AND/OR search tree [9, 11] framework is well suited for this kind of decomposable problem. The

Algorithm 1: AND/OR formulation with cache

```
1  Method search(currProblem: ∈ 𝒩^𝒟):(Tree,Cost)
2      return ORnode(currProblem, ∞)
3  Method ORnode(currProblem: ∈ 𝒩^𝒟, cost_ub):(Tree,Cost)
4      prefix_hash ← getPrefixHash(currProblem)
5      if storage.contains(prefix_hash) then      // optimal already computed
6          (sol_best, cost_best) ← storage.get(prefix_hash)
7          return (sol_best, cost_best)
8      else
9          cost_best ← cost_ub
10         sol_best ← null
11         forall f ∈ dom(d[currProblem]) do      // following value ordering
12             d[currProblem].assign(f)
13             if currProblem ∈ 𝒩^𝒩 then
14                 (sol_tree, cost_tree) ← ANDnode(currProblem, cost_best, f)
15             else
16                 sol_tree ← Tree(featureID : f left : null right : null)
17                 cost_tree ← e[currProblem]
18             if cost_best > cost_tree then
19                 cost_best ← cost_tree
20                 sol_best ← sol_tree
21         storage.add(prefix_hash, (sol_best, cost_best))     // new sol cached
22         return (sol_best, cost_best)
23 Method ANDnode(currProblem: ∈ 𝒩^𝒟, cost_ub, f_root):(Tree,Cost)
24     (sol_left, cost_left) ← ORnode(left(currProblem), cost_ub)        // 1st
25     if cost_left > cost_ub then
26         return (null, ∞)                  // pruning based on cost
27     (sol_right, cost_right) ← ORnode(right(currProblem), cost_ub − cost_left) // 2nd
28     sol_tree ← Tree(featureID : f_root left : sol_left right : sol_right)
29     return (sol_tree, cost_left + cost_right)
```

search is composed of two types of search nodes: the OR nodes (line 3) and the AND nodes (line 23). An example of the search tree for a decision tree of depth 2 is shown at Fig. 4.

The AND node is responsible for computing the optimal value of the left child (line 24), then the right child (line 27), and finally returns the composed solution (line 28). The OR node tests all the possible values for the root decision variable of `currProblem` (line 11). The ordering used to select the next value to test follows the principle of entropy [7]. The entropy of a set of transactions is computed using the number of transactions from each class, and is a well-known

heuristic in standard algorithms for learning decision trees:

$$Entropy(S) = -\frac{|\{t \in S : v[t] = 1\}|}{|S|} \log_2 \left(\frac{|\{t \in S : v[t] = 1\}|}{|S|} \right) - \frac{|\{t \in S : v[t] = 0\}|}{|S|} \log_2 \left(\frac{|\{t \in S : v[t] = 0\}|}{|S|} \right) \quad (29)$$

The information gain of a feature f is the difference between the initial entropy and the weighted entropy of a partition of the database into transactions with and without the feature:

$$Gain(f) = Entropy(D) - \frac{|\{t \in D : D_{t,f} = 1\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 1\}) - \frac{|\{t \in D : D_{t,f} = 0\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 0\}). \quad (30)$$

The classification is expected to be better when the gain is higher. We sort the values by decreasing gain. This ordering is computed once at the beginning of the search and is reused at every search node. After assigning the selected value, if the subtree still contains decision variables (i.e., if `currProblem` belongs to $\mathcal{N}^{\mathcal{N}}$), then the optimal subtrees are computed using an AND node (line 13). In the other case (i.e., if `currProblem` belongs to $\mathcal{N}^{\mathcal{E}}$), then we have already an optimal subtree (line 15). From all the values tested, the best sub-tree is kept (line 18) and returned (line 22).

Subtree equality. Two subproblems are equivalent whenever the set of decisions on the paths towards these nodes (the itemsets corresponding to the sets of decisions) are identical. Figure 3b shows how some subtrees can be the same in two different solutions due to paths that represent the same itemset. This is taken care of by using a caching system similar to the one used in the DL8 dynamic programming approach [13]. Two subtrees are equivalent if they share the same assigned prefix. The prefix of node i is composed of the values assigned to the decisions of the ancestors. These values are separated in two distinct sets: The *take* set $\{d[j] \mid j \in \text{ancestors}(i) \wedge \text{left}(j) \in \text{ancestors}(i) \cup \{i\}\}$, and the *drop* set $\{d[j] \mid j \in \text{ancestors}(i) \wedge \text{right}(j) \in \text{ancestors}(i) \cup \{i\}\}$. Two subtrees with the same *take* and *drop* sets are thus equivalent. A hash is computed from these sets and serves as key to store and retrieve the optimal subtree from storage (hashMap). In addition to the decision in the root of the subtree, its cost is also stored, easing the computation. The search for an already computed solution happens at the beginning of an OR node (line 5). A new solution is stored when a new complete optimal subtree is computed, i.e. at the end of the OR node (line 21).

Minimization. In order to decrease the number of explored search nodes, a pruning by minimization is added to the search. At each of the search nodes,

the upper bound of the allowed cost is propagated from node to node. During an OR node, this upper bound is decreased each time a better solution is found (line 18). During an AND node, the propagated upper bound is first propagated to the computation of the first child. If the result of this first child is above this propagated upper bound, then there is no need to compute the right child since any solution would be above the propagated upper bound (line 25). This is triggered if the best solution was already cached and has a higher cost than the bound or if there is no solution with a cost smaller than the upper bound. An invalid subtree is then returned. If the first child is lower than the upper bound, the second child can be computed and the propagated upper bound for its computation is the difference between the propagated upper bound of the tree and the cost of the already computed tree (line 27).

In practice. Oscar, the solver used in our experiment does not implement the AND/OR search tree framework. To avoid an invasive modification of the solver, the AND/OR tree is simulated using a custom OR tree. For further implementation details, the source code is available online[2].

5 Results

We compared our algorithm to two exact methods developed in earlier studies: BinOCT [19] and DL8 [13]. As both studies have already tested the quality of the resulting trees and as [3] had already stated that the more close to the optimal the tree is, the more accurate the classification, we decided to focus our experiments on the run time performance of our algorithm, and not on the validation of the quality of the trees.

Dataset	n	n^+	n^-	m
anneal	812	625	187	93
audiology	216	57	159	148
australian-credit	653	357	296	125
breast-wisconsin	683	444	239	120
diabetes	768	500	268	112
german-credit	1000	700	300	112
heart-cleveland	296	160	136	95
hepatitis	137	111	26	68
hypothyroid	3247	2970	277	88
ionosphere	351	225	126	445
kr-vs-kp	3196	1669	1527	73
letter	20000	813	19187	224

Dataset	n	n^+	n^-	m
lymph	148	81	67	68
mushroom	8124	4208	3916	119
pendigits	7494	780	6714	216
primary-tumor	336	82	254	31
segment	2310	330	1980	235
soybean	630	92	538	50
splice-1	3190	1655	1535	287
tic-tac-toe	958	626	332	27
vehicle	846	218	628	252
vote	435	267	168	48
yeast	1484	463	1021	89
zoo-1	101	41	60	36

Table 1: Description of the instances

[2] `https://bitbucket.org/helene_verhaeghe/classificationtree`

Dataset	d	$N_{\min}=1$						$N_{\min}=5$							
		DL8		BinOCT		CP		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t	obj	t	obj	t	obj	t
anneal	2	**137***	1	**137***	206	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**
anneal	3	**112***	37	**112**	TO	**112***	**6**	**112***	31	**112***	**7**	**112***	**7**	**112***	8
anneal	4	∞	TO	121	TO	**91***	**372**	**94***	591	**94***	365	**94***	508	**94***	**283**
anneal	5	∞	TO	120	TO	**86**	TO	∞	TO	**92**	TO	**92**	TO	**92**	TO
audiology	2	**10***	**< 1**	**10***	60	**10***	**< 1**	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**
audiology	3	**5***	62	7	TO	**5***	**15**	**7***	**2**	**7***	3	**7***	**2**	**7***	3
audiology	4	∞	TO	**1**	TO	**1**	TO	**4***	**43**	**4***	111	**4***	100	**4***	66
audiology	5	∞	TO	4	TO	**0**	TO	**1***	512	**1**	TO	**1**	TO	**1**	TO
australian-credit	2	**87***	2	**87***	206	**87***	**< 1**	**87***	2	**87***	**< 1**	**87***	**< 1**	**87***	1
australian-credit	3	**73***	124	86	TO	**73***	**29**	**74***	90	**74***	**27**	**74***	33	**74***	33
australian-credit	4	∞	TO	85	TO	**60**	TO	∞	TO	**66**	TO	**66**	TO	**66**	TO
breast-wisconsin	2	**22***	2	**22***	44	**22***	**< 1**	**22***	3	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**
breast-wisconsin	3	**15***	103	16	TO	**15***	**16**	**15***	80	**15***	18	**15***	**17**	**15***	23
breast-wisconsin	4	∞	TO	15	TO	**8**	TO	∞	TO	**10**	TO	**10**	TO	11	TO
diabetes	2	**177***	1	180	TO	**177***	**< 1**	**177***	1	**177***	**< 1**	**177***	**< 1**	**177***	**< 1**
diabetes	3	**162***	93	171	TO	**162***	**24**	**162***	90	**162***	**25**	**162***	32	**162***	29
diabetes	4	∞	TO	169	TO	**137**	TO	∞	TO	**138**	TO	**138**	TO	**138**	TO
german-credit	2	**267***	2	**267**	TO	**267***	**< 1**	**267***	2	**267***	**< 1**	**267***	**< 1**	**267***	**< 1**
german-credit	3	**236***	129	249	TO	**236***	**28**	**236***	122	**236***	**28**	**236***	37	**236***	33
german-credit	4	∞	TO	244	TO	**204**	TO	∞	TO	**205**	TO	**205**	TO	**205**	TO
heart-cleveland	2	**60***	**< 1**	**60***	312	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**
heart-cleveland	3	**41***	17	43	TO	**41***	**9**	**41***	15	**41***	**9**	**41***	11	**41***	11
heart-cleveland	4	**25***	**515**	39	TO	**25**	TO	**27***	**404**	**27**	TO	**27**	TO	**27***	497
heart-cleveland	5	∞	TO	34	TO	**10**	TO	∞	TO	**18**	TO	36	TO	**18**	TO
hepatitis	2	**16***	**< 1**	**16***	8	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**
hepatitis	3	**10***	4	12	TO	**10***	**2**	**11***	**2**	**11***	3	**11***	3	**11***	3
hepatitis	4	**3***	**54**	10	TO	**3***	92	**8***	**36**	**8***	98	**8***	112	**8***	69
hepatitis	5	∞	TO	7	TO	**0***	**19**	**5***	**299**	8	TO	8	TO	6	TO
hypothyroid	2	**70***	4	**70***	178	**70***	**< 1**	**70***	3	**70***	**< 1**	**70***	**< 1**	**70***	**< 1**
hypothyroid	3	**61***	122	62	TO	**61***	**12**	**62***	95	**62***	**11**	**62***	**11**	**62***	16
hypothyroid	4	∞	TO	62	TO	**53**	TO	∞	TO	**54**	TO	**54**	TO	**54***	**552**
ionosphere	2	**32***	50	**32**	TO	**32***	**4**	**32***	48	**32***	**4**	**32***	**4**	**32***	8
ionosphere	3	∞	TO	29	TO	**22**	TO	∞	TO	**22**	TO	**22**	TO	24	TO
ionosphere	4	∞	TO	26	TO	**14**	TO	∞	TO	∞	TO	**20**	TO	∞	TO
kr-vs-kp	2	**418***	2	418	TO	**418***	**< 1**	**418***	2	**418***	**< 1**	**418***	**< 1**	**418***	**< 1**
kr-vs-kp	3	**198***	74	301	TO	**198***	**6**	**198***	63	**198***	**8**	**198***	**8**	**198***	11
kr-vs-kp	4	∞	TO	877	TO	**144***	**378**	∞	TO	**144***	455	**144***	554	**144***	**345**
kr-vs-kp	5	∞	TO	675	TO	**132**	TO	∞	TO	**132**	TO	**132**	TO	**132**	TO

Table 2: Results (part 1) Time out = 10 min, best value (obj or time) for a given $N_{\min}$ in bold, optimal obj proven indicated with $*$

The benchmark is composed of instances from the CP4IM[3] and UCI [4] websites. Their description is given at Table 1. BinOCT is a MIP-based approach running on CPLEX. It does not allow to give a specific value for $N_{\min}$. If a timeout is reached, the method outputs its best solution so far. We used the implementation available online with as arguments the depth, the timeout (10 min) and a polishing time (2.5 min). The polishing time is used to configure the CPLEX solver. At timeout minus the polishing time, CPLEX changes its search strategy. Polishing [17] is time consuming, but it allows improving a solution when the search stagnates. DL8 is a dynamic programming approach. It com-

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

[4] `https://archive.ics.uci.edu/ml/index.php`

Dataset	d	$N_{\min}=1$						$N_{\min}=5$							
		DL8		BinOCT		CP		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t	obj	t	obj	t	obj	t
letter	2	∞	TO	813	TO	**599***	**9**	∞	TO	**599***	**13**	**599***	**13**	**599***	24
letter	3	∞	TO	813	TO	**531**	TO	∞	TO	**531**	TO	**531**	TO	532	TO
lymph	2	**22***	**< 1**	**22***	17	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**
lymph	3	**12***	**2**	13	TO	**12***	**2**	**13***	**1**	**13***	2	**13***	2	**13***	2
lymph	4	**3***	**43**	8	TO	**3***	85	**7***	**15**	**7***	54	**7***	58	**7***	44
lymph	5	∞	TO	8	TO	**0***	**5**	**4***	166	**4**	TO	**4**	TO	**4**	TO
mushroom	2	**252***	27	520	TO	**252***	**< 1**	**252***	24	**252***	**< 1**	**252***	1	**252***	1
mushroom	3	∞	TO	396	TO	**8***	**36**	∞	TO	**8***	**46**	**8***	**46**	**8***	64
mushroom	4	∞	TO	160	TO	**0***	**< 1**	∞	TO	**0***	**< 1**	**0***	**< 1**	**0**	TO
pendigits	2	∞	TO	**153**	TO	**153***	**3**	∞	TO	**153***	**4**	**153***	**4**	**153***	8
pendigits	3	∞	TO	496	TO	**47**	TO	∞	TO	**47**	TO	**47**	TO	**47**	TO
primary-tumor	2	**58***	**< 1**	**58***	5	**58***	**< 1**	**58***	**< 1**	**58***	**< 1**	**58***	**< 1**	**58***	**< 1**
primary-tumor	3	**46***	**< 1**	49	TO	**46***	**< 1**	**46***	**< 1**	**46***	**< 1**	**46***	**< 1**	**46***	**< 1**
primary-tumor	4	**34***	**2**	39	TO	**34***	6	**40***	**1**	**40***	6	**40***	9	**40***	5
primary-tumor	5	**26***	**14**	37	TO	**26***	129	**34***	**8**	**34***	103	**34***	224	**34***	39
segment	2	**9***	49	**9**	TO	**9***	**1**	**9***	41	**9***	**2**	**9***	**2**	**9***	3
segment	3	∞	TO	6	TO	**0***	**13**	∞	TO	**2***	**242**	**2***	248	**2***	399
segment	4	∞	TO	21	TO	**0***	**115**	∞	TO	**0**	TO	**0**	TO	1	TO
soybean	2	**55***	**< 1**	**55***	19	**55***	**< 1**	**55***	**< 1**	**55***	**< 1**	**55***	**< 1**	**55***	**< 1**
soybean	3	**29***	2	42	TO	**29***	**1**	**29***	2	**29***	**1**	**29***	**1**	**29***	2
soybean	4	**14***	**33**	16	TO	**14***	34	**15***	27	**15***	37	**15***	44	**15***	**26**
soybean	5	**8***	**315**	24	TO	**8**	TO	**13***	**239**	**13**	TO	**13**	TO	**13***	407
splice-1	2	**508***	143	522	TO	**508***	**5**	**508***	89	**508***	**4**	**508***	**4**	**508***	7
splice-1	3	∞	TO	574	TO	**224**	TO	∞	TO	**225**	TO	**225**	TO	**225**	TO
tic-tac-toe	2	**282***	**< 1**	**282***	10	**282***	**< 1**	**282***	**< 1**	**282***	**< 1**	**282***	**< 1**	**282***	**< 1**
tic-tac-toe	3	**216***	**< 1**	231	TO	**216***	**< 1**	**216***	**< 1**	**216***	**< 1**	**216***	1	**216***	**< 1**
tic-tac-toe	4	**137***	**3**	169	TO	**137***	7	**137***	**3**	**137***	7	**137***	15	**137***	6
tic-tac-toe	5	**63***	**16**	128	TO	**63***	125	**63***	**16**	**63***	140	**63***	435	**63***	63
vehicle	2	**75***	23	**75**	TO	**75***	**1**	**75***	20	**75***	**1**	**75***	**1**	**75***	3
vehicle	3	∞	TO	60	TO	**26***	**236**	∞	TO	**28***	**266**	**28***	292	**28***	473
vehicle	4	∞	TO	84	TO	**19**	TO	∞	TO	**21**	TO	**21**	TO	**21**	TO
vote	2	**17***	**< 1**	**17***	8	**17***	**< 1**	**18***	**< 1**	**18***	**< 1**	**18***	**< 1**	**18***	**< 1**
vote	3	**12***	2	13	TO	**12***	**1**	**13***	**1**	**13***	**1**	**13***	**1**	**13***	2
vote	4	**5***	**23**	11	TO	**5***	44	**6***	**13**	**6***	29	**6***	33	**6***	29
vote	5	**1***	**248**	5	TO	**1**	TO	**3***	**118**	**3***	430	**3***	496	**3***	417
yeast	2	**437***	2	**437**	TO	**437***	**< 1**	**437***	2	**437***	**< 1**	**437***	**< 1**	**437***	**< 1**
yeast	3	**403***	74	430	TO	**403***	**12**	**403***	70	**403***	**15**	**403***	19	**403***	17
yeast	4	∞	TO	412	TO	**367**	TO	∞	TO	**367**	TO	**367**	TO	**367**	TO
zoo-1	2	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**

Table 3: Results (part 2) Time out = 10 min, best value (obj or time) for a given $N_{\min}$ in bold, optimal obj proven indicated with $*$

	$N_{\min}=1$			$N_{\min}=5$			
	DL8	BinOCT	CP	DL8	CP	CP-c	CP-m
Proven optimality	49(64%)	13(17%)	**57**(75%)	54(71%)	56(74%)	56(74%)	**58**(76%)
Best solution found	49(64%)	21(28%)	**76**(100%)	54(71%)	**74**(97%)	**74**(97%)	70(92%)
Fastest	23(30%)	11(14%)	**49**(64%)	28(37%)	**40**(53%)	33(43%)	22(29%)
Time out	27(36%)	63(83%)	**19**(25%)	22(29%)	21(28%)	21(28%)	**19**(25%)

Table 4: Summary of the results

putes a subset of the frequent itemsets and then builds the optimal tree from it. This approach does not output any intermediate non-optimal tree. We used the

implementation provided by the authors with as arguments the depth and the minimum support (value of $N_{\min}$).

The first part of Table 2 and Table 3 shows the results for the three methods (DL8, BinOCT and ours) with $N_{\min} = 1$ using a timeout of 10 mins. The second part of Table 2 and Table 3 shows the results for two methods (DL8 and ours) and some variations of our approach (without the caching and without the pruning using bounds) with $N_{\min} = 5$ using a timeout of 10 mins. This comparison does not include BinOCT since its implementation cannot take into account $N_{\min}$. A value of 5 is chosen, as this yields results that are more statistically significant. Table 4 summarizes our results. For each of the algorithms, the number of instances where the optimality is proven, the solution found is the best among the tested algorithms, the algorithm was the fastest and timeout is reached are gathered.

Our method outperforms the two others on most of the instances. It could find and prove optimality on roughly 75% of the instances within the time limit. The best solution found was reached by our method in almost every cases. However, DL8 performs better on small instances such as *hepatitis*, *lymph* or *primary-tumor*. The large difference between BinOCT and our method can be explained by the benefits of the AND/OR search that is not used by BinOCT. The gap with DL8 can be partially explained by the cost pruning. It can possibly also be explained by the itemset mining algorithms used: DL8 lacks the optimizations found in the CoverSize constraint [18].

Finally the effects of the cache and the pruning using the best known partial solutions can be observed. CP-c gives the results of our method when the cache system is not used and CP-m gives the results when the pruning using the best partial solution is not used. The cache becomes really useful at depth 4 (or more) and some instances greatly benefit from it (e.g. the *anneal* benchmark with a depth of 4 improves its timing by almost 30% when adding the cache). The effect of the pruning is significant in some cases. On some benchmarks such as *mushroom*, *ionosphere* or *vehicle*, the pruning improves greatly the solution (ex. on *vehicle* depth 4, the time is divided by 1.7). On other benchmarks such as *hepatitis*, *lymph* or *primary-tumor*, it decreases the performance. These instances coincide with the ones where DL8 performs better.

6 Conclusion

We presented a new approach for efficiently creating an optimal decision tree of limited depth. On most of the benchmarks, it gives the best solution within the allocated time and is the fastest to prove optimality.

We believe our approach can be extended in a number of different ways. It is straightforward to extend it to the multiclass setting, by adding counters and CoverSizeSR constraints for each of the additional classes. We assumed the input data was binary; if the data was not binary, it can be binarized beforehand [6]. Of particular interest can also be addition of further constraints and the use of other cost functions that can be expressed as a sum of costs at the leaves.

References

1. Aghaei, S., Azizi, M.J., Vayanos, P.: Learning optimal and fair decision trees for non-discriminative decision-making (2019)
2. Babaki, B., Guns, T., De Raedt, L.: Stochastic constraint programming with and-or branch-and-bound. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 539–545 (2017)
3. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning **106**(7), 1039–1082 (2017)
4. Bessiere, C., Hebrard, E., O'Sullivan, B.: Minimising decision tree size as combinatorial optimisation. In: International Conference on Principles and Practice of Constraint Programming. pp. 173–187. Springer (2009)
5. Bonfietti, A., Lombardi, M., Milano, M.: Embedding decision trees and random forests in constraint programming. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 74–90. Springer (2015)
6. Breiman, L.: Classification and regression trees. Routledge (1984)
7. Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2012)
8. Dechter, R., Mateescu, R.: The impact of AND/OR search spaces on constraint satisfaction and counting. In: Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings. pp. 731–736 (2004)
9. Dechter, R., Mateescu, R.: And/or search spaces for graphical models. Artificial intelligence **171**(2-3), 73–106 (2007)
10. Laurent, H., Rivest, R.L.: Constructing optimal binary decision trees is np-complete. Information processing letters **5**(1), 15–17 (1976)
11. Marinescu, R., Dechter, R.: And/or tree search for constraint optimization. In: Proc. of the 6th International Workshop on Preferences and Soft Constraints. Citeseer (2004)
12. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J., RAS, I.: Learning optimal decision trees with sat. In: IJCAI. pp. 1362–1368 (2018)
13. Nijssen, S., Fromont, E.: Mining optimal decision trees from itemset lattices. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 530–539. ACM (2007)
14. Nijssen, S., Fromont, É.: Optimal constraint-based decision tree induction from itemset lattices. Data Min. Knowl. Discov. **21**(1), 9–51 (2010)
15. Quinlan, J.R.: Induction of decision trees. Machine learning **1**(1), 81–106 (1986)
16. Quinlan, J.R.: C4. 5: programs for machine learning. Elsevier (1993)
17. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
18. Schaus, P., Aoga, J.O., Guns, T.: Coversize: a global constraint for frequency-based itemset mining. In: International Conference on Principles and Practice of Constraint Programming. pp. 529–546. Springer (2017)
19. Verwer, S., Zhang, Y.: Learning optimal classification trees using a binary linear program formulation. In: 33rd AAAI Conference on Artificial Intelligence (2019)