Apprentissage d'arbres de décision optimaux grâce à la programmation par contraintes

Hélène Verhaeghe[1*] **Siegfried Nijssen**[1] **Gilles Pesant**[2]
Claude-Guy Quimper[3] **Pierre Schaus**[1]
[1] UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium
[2] Polytechnique Montréal, Montréal, Canada
[3] Université Laval, Québec, Canada
[1]{prenom.nom}@uclouvain.be [2]gilles.pesant@polymtl.ca
[3]claude-guy.quimper@ift.ulaval.ca

Résumé

Les arbres de décision sont parmi les modèles les plus populaires en apprentissage automatique. L'utilisation d'algorithmes gloutons pour les apprendre peut poser plusieurs désavantages : il est difficile de limiter la taille de l'arbre de décision tout en maintenant une bonne qualité de classification, et il est difficile d'imposer de nouvelles contraintes sur le modèle appris. Ces raisons sont à la base de l'émergence d'un intérêt pour des algorithmes exacts et flexibles dans l'apprentissage des arbres de décision. Ce papier résume notre papier "Learning Optimal Decision Trees using contraint Programming", accepté en version journal accéléré à CP2019 [5]. Dans ce papier, nous introduisons une nouvelle approche pour apprendre des arbres de décision en utilisant la programmation par contraintes.

1 Introduction

Les arbres de décision sont un modèle de classification populaire en apprentissage automatique. Leurs bénéfices incluent leur relative facile interprétation et le fait qu'ils donnent de bonnes performances de classification sûr de nombreux jeux de données.

De nombreuses méthodes ont été proposées dans la littérature pour apprendre des arbres de décision.Les méthodes gloutonnes sont les plus populaires.Ces méthodes partitionnent de manière récursive le jeu de données en deux sous-ensembles. Cette partition se base sur un attribut, sélectionné de manière gloutonne. Le partitionnement récursif s'arrête lorsque le critère d'arrêt est atteint (par exemple, un nombre minimum d'éléments du jeu de données initiales dans une feuille ou une classe unique pour ces éléments).Bien qu'en pratique ces méthodes arrivent à une bonne précision de classification pour beaucoup de types de données, malheureusement, elles ne donnent que peu de garantie.Résultant de cela, les arbres obtenus par ces méthodes peuvent être inutilement complexes, moins précis que possibles et il est également compliqué d'imposer de nouvelles contraintes sur ces arbres.

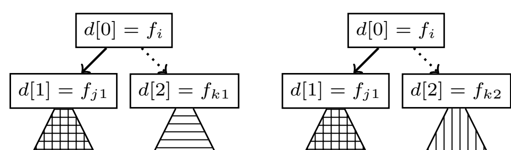
Pour pallier ces faiblesses, des chercheurs ont étudié l'inférence d'arbres de décision optimaux sous contraintes [3, 1]. Ces approches assurent que sous certaines contraintes bien définies et considérant un critère d'optimisation prédéfini, un arbre optimal est trouvé.

Notre papier propose une nouvelle approche, plus évolutive, basée sur la programmation par contraintes (PPC) pour apprendre des arbres de décision avec une profondeur maximale fixée qui minimise l'erreur de classification. Notre approche combine les idées clés suivantes : l'utilisation des mécanismes d'un solveur PPC, l'utilisation de la contrainte globale CoverSize [4], l'utilisation d'un arbre de recherche ET/OU [2] et l'utilisation d'un mécanisme de mémoization tel qu'introduit par DL8 [3]. Ces éléments permettent à notre approche par programmation par contraintes de profiter, de manière plus efficace, de la décomposition de ce problème d'apprentissage. Nous allons montrer que la combinaison de ces différentes idées mène à un modèle qui est plus efficace que les autres approches proposées dans la littérature.

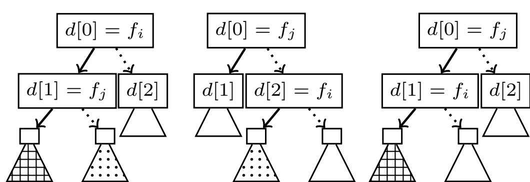
*Papier doctorant : Hélène Verhaeghe[1] est auteur principal.

2 Modèle et arbre de recherche

Les variables de décisions principales nécessaires pour modéliser notre arbre de décision sont les décisions prises à chaque noeud de l'arbre. Un autre ensemble de variables est utilisé pour compter le nombre de transaction correspondant à chaque feuille. L'intégrité de l'arbre est assurée par l'utilisation de contraintes AllDifferent, de contraintes CoverSize et de simples contraintes arithmétiques liant les différentes variables. D'autres contraintes redondante sont ajoutée pour accélérer la résolution.



(a) Indépendance des sous-arbres



(b) Sous-arbres redondants ; Redundant subtrees ; les sous-arbres mis en évidence de la même manière sont identiques

Figure 1 – Décompositions

Les solveurs traditionnels de PPC utilisent un arbre OU. Dans ceux-ci, pour chaque noeud, la solution se trouve dans l'une des branches. Pour trouver l'arbre de décision optimal, nous utilisons un arbre ET/OU. Un noeud ET va diviser le problème en sous problèmes indépendants. La solution est donc la combinaison des solutions des sous problèmes. Les noeuds OU sont les mêmes que classiquement utilisés. L'utilisation d'un tel arbre est rendue possible par le caractère indépendant de deux sous-arbres de décision pour lesquels la racine a été fixée (Fig.1a).

De plus, un mécanisme de mémoization permet le stockage de sous-arbres optimaux. En effet, si les ensembles des décisions pour arriver à deux sous-arbres sont équivalents, alors les sous-arbres optimaux sont les identiques (Fig.1b).

3 Résultats

Nous avons comparé notre méthode à DL8 [3] et BinOCT [6], deux méthodes résolvant le même problème. Notre méthode surpasse les deux autres sur la plupart des instances. Elle a pu prouver l'optimalité sur environ 80% des instances endéans la limite de temps. La meilleure solution trouvée est obtenue par notre méthode dans près de tous les cas. Cependant, DL8 performe mieux sur les petites instances. La grande différence entre BinOCT et notre méthode peut être expliquée par le bénéfice de la recherche ET/OU qui n'est pas utilisé par BinOCT. L'écart avec DL8 peut être partiellement expliqué par l'élagage du coût. Cela peut également venir de l'algorithme de recherche d'ensemble fréquent utilisé : DL8 ne profite pas des optimisations présentes dans la contrainte CoverSize. Nos expérimentations évaluent également les effets de certaines des techniques utilisées pour résoudre notre problème, tel que l'utilisation de la mémoization.

4 Conclusion

Pour résumer, notre papier présente une nouvelle approche pour créer de manière efficace un arbre de décision optimal avec profondeur limitée. Cette approche basée sur la PPC combine la contrainte globale CoverSize, le concept d'arbre de recherche ET/OU et la mémoization. Sur la plupart des jeux de données, notre algorithme trouve la meilleure solution dans le temps impartit et est le plus rapide à prouver l'optimalité. Nous pensons que notre approche peut être étendue de plusieurs manières (multi-classe, caractéristiques continues en utilisant la binarisation,...).

Références

[1] Gaël Aglin, Siegfried Nijssen et Pierre Schaus : Learning optimal decision trees using caching branch-and-bound search. *In AAAI20*, 2020.

[2] R. Dechter et R. Mateescu : And/or search spaces for graphical models. *Artificial intelligence*, 171(2-3):73–106, 2007.

[3] S. Nijssen et E. Fromont : Mining optimal decision trees from itemset lattices. *In SIGKDD2007*, 2007.

[4] P. Schaus, J. Aoga et T. Guns : Coversize : a global constraint for frequency-based itemset mining. *In International Conference on Principles and Practice of Constraint Programming*, pages 529–546. Springer, 2017.

[5] Hélene Verhaeghe, Siegfried Nijssen, Gilles Pesant, Claude-Guy Quimper et Pierre Schaus : Learning optimal decision trees using constraint programming. *Constraints*, 25(3):226–250, 2020.

[6] S. Verwer et Y. Zhang : Learning optimal classification trees using a binary linear program formulation. *In AAAI19*, 2019.

Noname manuscript No.
(will be inserted by the editor)

Learning Optimal Decision Trees using Constraint Programming

Hélène Verhaeghe · Siegfried Nijssen · Gilles Pesant · Claude-Guy Quimper · Pierre Schaus

the date of receipt and acceptance should be inserted later

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, Caching

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, 19, 20]. These methods

H. Verhaeghe, S. Nijssen and P. Schaus
UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium, E-mail: {firstname.lastname}@uclouvain.be

G.Pesant
Polytechnique Montréal, Montréal, Canada, E-mail: gilles.pesant@polymtl.ca

C.-G. Quimper
Université Laval, Québec, Canada, E-mail: claude-guy.quimper@ift.ulaval.ca

recursively partition a dataset into two subsets based on a greedily selected attribute until some stopping criterion is reached (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 [1,3,4,15–17,26][1]. These approaches ensure that under well-defined constraints and optimization criteria, an optimal tree is found. Experiments conducted in earlier work [3,16,17] 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 [3,16,17,26] 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 leaves of the tree [3,17], on fairness [1], or on the preservation of privacy by these trees [17].

We suspect, as papers [3,17,26], that the problem of finding an optimal decision tree given a maximum depth is NP-complete even if no formal proof is available yet. Clearly the problem is in NP. As the depth is bounded, at most d tests are required to classify one of the transactions from the database. Therefore, the error can be computed in $\mathcal{O}(nd)$ (with n the total number of transactions and d the maximum depth), which is polynomial. Similar problems have been proven NP-Complete. Hyafil and Rivest [12] found that the problem of finding a decision tree which minimizes the expected number of tests to classify a new transaction is NP-complete. The problem of finding a decision tree minimizing the number of leaves is also proven NP-complete in [11].

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 leaf examples end up [23];

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

- 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 [16].

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 5. Finally, Section 6 presents empirical results about our algorithm.

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 [1,3,26] 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 [17]. 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 [4,15] 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 [16, 26].

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. $D_{i,j}$ represents the value of the jth feature of D_i, i.e., 0 or 1 as we work with binary features. We assume in this work that each transaction belongs to one of two classes, represented by 0 and 1. The classes are stored in a vector v of size n. 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

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

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

We focus on creating well-formed decision trees. This means that a given feature can be used at most once on each path from the root to a leaf. Also, each subtree rooted in one of the inner nodes should not have all its leaves assigned to the same class.

We define the *depth* d 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 the class of the transaction, $v[t]$, is different from the classification by the tree, $F(r, t)$):

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

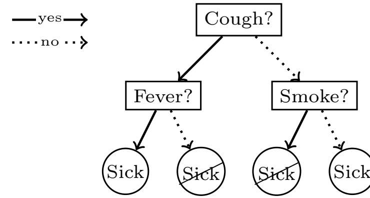
We allow for the additional specification of a constraint on the minimum number of examples $N_{\min}$ in each leaf of the tree [3, 17],

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.

Let us give a simple example. Given the dataset in Fig. 1a, a possible decision tree of depth 2 is provided in Fig. 1b. For patient 1, the tree predicts the correct class (she does not have a cough and does not smoke, which leads to the fourth leaf, which categorizes the patient as sick). For patient 4, the tree produces a

	Fever?	Cough?	Age > 60?	Smoke?	...	Sick from A?
Patient 1	✓	✘	✓	✘	...	✓
Patient 2	✓	✓	✘	✘	...	✓
Patient 3	✘	✘	✓	✓	...	✘
Patient 4	✘	✓	✓	✓	...	✓
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) Dataset



(b) Decision tree

Fig. 1: Example

faulty classification (she has a cough and does not have a fever, which leads to the second leaf and categorizes the patient as not sick). The tree is the optimal decision tree of maximum depth 2 with respect to the classification error if the number of misclassified patients is the smallest among all the possible trees of maximum depth 2.

3.2 Constraint Programming

Constraint programming (CP) [21] is a computational paradigm aiming at solving combinatorial problems (satisfaction and optimization ones). The problem is stated as a set of *variables* and a set of *constraints* acting on the variables. The constraints represent the properties that should be respected in a solution (i.e. an assignment of a value to each variable). An example of a constraint is the *AllDifferent* constraint. It specifies that each variable involved should take a different value in a valid assignment. The variables and constraints form the *model* of the problem. A *search tree* is then explored (usually using backtracking depth first search) to discover an assignment of all the variables that satisfies all the constraints. A heuristic decides at each node of the search tree which variable and value should be assigned/removed in the next two alternative branches. For example, a first fail heuristic selects, at each node of the search tree, the unbound variable with the smallest remaining domain. The search tree is pruned by the constraints in charge of removing infeasible values from the domain of the variables during the fix-point computation triggered at each node of the search tree. A backtrack occurs when one domain becomes empty. We refer to [13] for more information about Constraint Programming technology.

3.3 AND/OR Search Trees

In a classical CP framework, the search tree is only composed of OR nodes. Each branch starting at an OR node adds a simple constraint in order to cut the search space. To keep the search complete, the branches should be complementary and together still represent the full search space. The optimal solution lies in one of the branches. An example could be to branch using $x = v$ and $x \neq v$. The first branch restricts the domain of x to the singleton $\{v\}$, while the other branch restricts it to $dom(x)\backslash\{v\}$. The union of the two branches covers all the possible values of x. Another example could be to create a branch for each of the values of the domain of a given variable, each assigning this value to the variable.

In AND/OR search trees [9, 14], a second type of node is introduced: the AND nodes. Each branch starting at an AND node represents a distinct, independent subproblem. The set of unbound variables and the set of remaining constraints are partitioned into disjoint sets. The solution, if it exists, is therefore the conjunction of the partial solutions of all the branches. If one branch does not have a solution, then no solution exists for the node. An example: given a problem of 5 variables ($dom(A) = \{5\}$, $dom(B_1) = dom(B_2) = dom(C_1) = dom(C_2) = \{0, 1, 2, 3, 4, 5\}$) and two constraints ($A = B_1 + C_1$ and $A = B_2 + C_2$), since A is bound, the problem can be partitioned into two independent subproblems. On the first branch, B_1, C_1 and $A = B_1 + C_1$ form the first sub-problem, while B_2, C_2 and $A = B_2 + C_2$ forms the second, on the second branch.

An AND/OR search tree combines both OR and AND nodes in order to find solutions to the problem.

3.4 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 [16, 17]. We exploit this correspondence by adapting the CoverSize global constraint [23] 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:

$$\textsc{CoverSize}([I_1, \ldots, I_m], D, c) \iff c = \left| \bigcap_{I_i = 1} \{t \in \{1, \ldots, n\} \mid D_{t,i} = 1\} \right|. \tag{3}$$

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 algorithm described in [23] is bound consistent.

For example, reusing the dataset in Fig. 1a, each Boolean is linked to one of the features (*Fever?*, *Cough?*,...). Given the selected features (i.e. corresponding Boolean set to true) forming the pattern, the constraint ensures that the counter corresponds to how many transactions matches this pattern. If the only true Boolean are $I_{Fever?}$ and $I_{Age>60?}$, then on the displayed patients, only the first should be counted, since she is the only one to have both features.

4 Preliminaries: Adaptation of CoverSize

The CoverSize constraint, in its original form, does not suit our application right away. Our aim is to use it to link the number of transactions matching a given series of decisions (i.e. the path to the corresponding leaf). Given a tree, we know two things about each series of decisions leading to a leaf. One, there is a fixed number of decisions along the path and second, some are also rejecting decisions. This led to the modifications made to the CoverSize propagation algorithm.

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

$$\textsc{CoverSizeS}(\{K_1, ..., K_a\}, D, c) \iff c = \left| \bigcap_{i=1}^{a} \{t \in \{1, ..., n\} \mid D_{t,K_i} = 1\} \right| \quad (4)$$

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.

Reusing the example in Fig. 1a and a pattern of size 2, if the two values assigned to the variables are *Fever?* and *Age>60?*, then the constraint should ensure that only patient 1 is counted (from the four displayed).

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:

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

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

The *take* (resp. *drop*) set defines the features that should (resp. should not) appear in the counted transactions.

Reusing the dataset in Fig. 1a, with a pattern consisting of two features, one in the take set and one in the drop set, if their values are respectively *Cough?* and *Fever?*, then only patient 4 (over the one displayed) matches the pattern and is therefore counted. In the decision tree of Fig. 1b, this pattern corresponds to the second leaf.

The pseudo-code of the CoverSizeSR propagator is given as Algorithm 1. Algorithm 2 details two methods used by the propagator. The key element of the algorithm is the cover. It represents the set of transactions corresponding to the features already selected in the take and drop sets. As in the original implementation of CoverSize, the cover is implemented using a reversible sparse bitset [10]. In this auto-backtracking structure, each bit is associated with one of the transactions of the database. If the transaction is still valid concerning the already selected features, then its associated bit is set to 1. It is set to 0 otherwise. To

help the computations, immutable bitsets are precomputed for each of the possible features. Each of these bitsets maintains the set of transactions containing the given feature.

Algorithm 1: PropagateCoverSizeSR

```
1  take: Set of variables                                  // set of take variable
2  drop: Set of variables                                  // set of drop variable
3  c: Variable                                             // counter variable
4  cover: Reversible Sparse Bitset                         // current cover
5  sizeCover: Integer                                      // size of current cover
6  support: Array of Bitset                                // precomputed bitsets
7  Function propagate(take, drop, c, cover, sizeCover, support)
8      vars = take ∪ drop
9      takeUnbound ← { x | x ∈ take ∧ |dom(x)| > 1 }
10     remainingTakeVals ← ⋃_{x∈takeUnbound} dom(x)
11     dropUnbound ← { x | x ∈ drop ∧ |dom(x)| > 1 }
12     remainingDropVals ← ⋃_{x∈dropUnbound} dom(x)
13     isCoverChanged ← updateCover(take, drop, cover, sizeCover, support)
        // method defined at Algo.2
14     c.max ← min(sizeCover, c.max)
15     if { x | x ∈ vars ∧ |dom(x)| > 1 } = ∅ then
16         c.min ← sizeCover
17     else
18         filterValues(remainingTakeVals, takeUnbound, remainingDropVals,
            dropUnbound, c, cover, support)          // method defined at Algo.2
           /* compute cover as if every decision available for the take set were
              selected in the cover and every decision available for the drop
              set were rejected from the cover                                  */
19         if isCoverChanged ∧ ( remainingTakeVals ∩ remainingDropVals = ∅ )
            then
20             virtualCover ← cover
21             foreach i ∈ remainingTakeVals do
22                 virtualCover ← virtualCover ∩ support[i]
23             foreach i ∈ remainingDropVals do
24                 virtualCover ← virtualCover ∩ support[i]^C
25             lb ← | virtualCover |
26             c.min ← max(lb, c.min)
           /* if the counter variable is bounded to the current size of the
              cover, remove all values that would reduce the size of the cover
              */
27         if |dom(c)| = 1 ∧ c.min = sizeCover then
28             foreach i ∈ remainingTakeVals do
29                 if cover ∩ support[i] ≠ cover then
30                     foreach x ∈ takeUnbound do
31                         dom(x) ← dom(x)\{i}
32             foreach i ∈ remainingDropVals do
33                 if cover ∩ support[i]^C ≠ cover then
34                     foreach x ∈ dropUnbound do
35                         dom(x) ← dom(x)\{i}
```

Algorithm 2: PropagateCoverSizeSR: functions updateCover and filterValues

```
1  Function updateCover(take, drop, cover, sizeCover, support)
2    mask ← cover
     /* Update cover with the new values chosen in the take set */
3    foreach x ∈ take do
4      if x newly bound then // bound since last propagation
5        mask ← mask ∩ support[x.value]
     /* Update cover with the new values rejected in the drop set */
6    foreach x ∈ drop do
7      if x newly bound then // bound since last propagation
8        mask ← mask ∩ support[x.value]^C
9    if cover ≠ mask then
10     cover ← mask
11     sizeCover ← |cover|                      // cover upper bound
12     return true
13   else
14     return false
15 Function filterValues(remainingTakeVals, takeUnbound, remainingDropVals,
    dropUnbound, c, cover, support)
     /* Test remaining values available for the take set */
16   foreach i ∈ remainingTakeVals do
17     count ← | cover ∩ support[i] |
18     if count < c.min then // too few left to select
19       foreach x ∈ takeUnbound do
20         dom(x) ← dom(x)\{i}
21       remainingTakeVals ← remainingTakeVals\{i}
     /* Test remaining values available for the drop set */
22   foreach i ∈ remainingDropVals do // Remove impossible values
23     count ← | cover ∩ support[i]^C |
24     if count < c.min then // too few left to reject
25       foreach x ∈ dropUnbound do
26         dom(x) ← dom(x)\{i}
27       remainingDropVals ← remainingDropVals\{i}
```

The algorithm first updates the cover (Algo.2 line 1) for each of the variables newly bound. For each variable from the take set (Algo.2 line 3), the current cover is intersected with the support of the feature. For each variable from the drop set (Algo.2 line 6), the current cover is intersected with the complement of the support of the feature. An updated cover allows to compute the new value of the upper bound of the counter.

If all the variables from both take and drop sets have been assigned, then the cover cannot evolve (Algo.1 line 15). The previously computed upper bound is also the lower bound. Otherwise, the computation continues.

Then, some features may be now impossible to select or reject and should be filtered out of the domains (Algo. 2 line 15). This is triggered by a change in the cover or a change in the domain of the counter. To test this, a virtual inclusion (resp. rejection) of the feature is done (Algo. 2 line 17 (resp. line 23)) by doing the intersection between the cover and the concerned support (resp. the complement of

the concerned support). This intersection corresponds to the number of remaining transactions with (resp. without) the feature. Using the size of this intersection, we can easily prevent the feature from being used in the take (resp. drop) set. If the size is smaller than the current minimum, then the feature cannot be assigned to a take (resp. drop) set variable, i.e. not enough transactions with (resp. without) the feature to meet the current lower bound of the counter.

The next step is to compute a new lower bound (Algo. 1 line 19). This is done by virtually selecting all the remaining values from both take and drop set into the cover. All the supports of the available values for the take set are intersected with the cover and the complement of the supports of the available values for the drop set are intersected. The size of this virtual cover is the smallest cover possible. However, if a given value is still allowed in both the take and drop set, by the property stating that the intersection of a set and its complement is always empty, the virtual cover is empty and the lower bound is equal to 0. The computation of the lower bound can thus be avoided in such cases.

Finally, if the counter ends up taking the value of the size of the cover, the features which modify the cover, and thus its size, should be removed from the domains (Algo. 1 line 27).

The time complexity of CoverSizeSR is $\mathcal{O}(m\frac{n}{w})$ with w the size of the computer words (e.g., $w = 64$). This is the same complexity as the CoverSize propagator. The space complexity is $\mathcal{O}(m\frac{n}{w})$. The consistancy remains the same: bound consistent.

5 CP Modeling of the Problem

5.1 Model of the Problem

In this section, we will introduce the variables and constraints used in our model. Fig. 2 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. Unlike other features used at most once per path, this feature is allowed multiple times to allow any tree shape. A node with this value therefore has no transactions from the database on its left branch. This assumption is also motivated by the CP framework. Each potentially used variable should be defined before the start of the search. Figure 3 shows how a proper tree can be made perfect by the use of the dummy feature.

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,

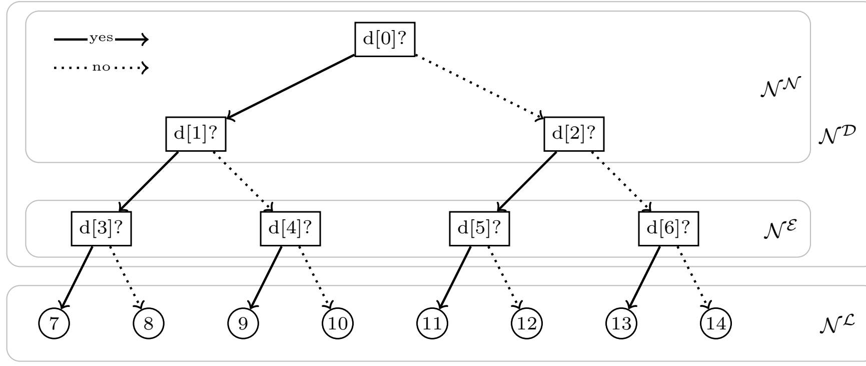
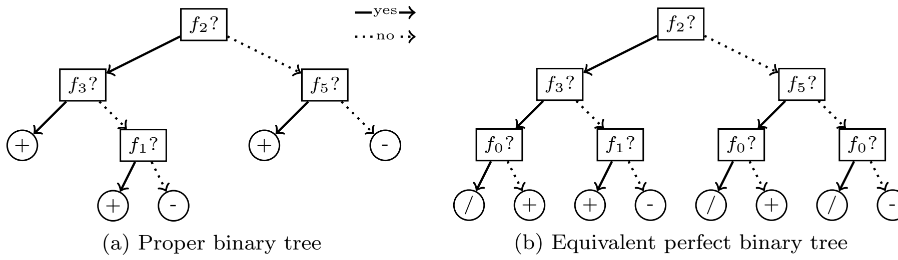


Fig. 2: Representation of a perfect decision tree of depth 3



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

Fig. 3: Example of the use of the dummy feature f_0 to transform the proper binary tree into a perfect binary tree

the number of variables and constraints only depends on the number of nodes in the tree.

5.1.1 Variables

In our model we have variables with the following domains:

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

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

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

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

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

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.

5.1.2 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).

5.1.3 Objective

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

5.1.4 Redundant constraints

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

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

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

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

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

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

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

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

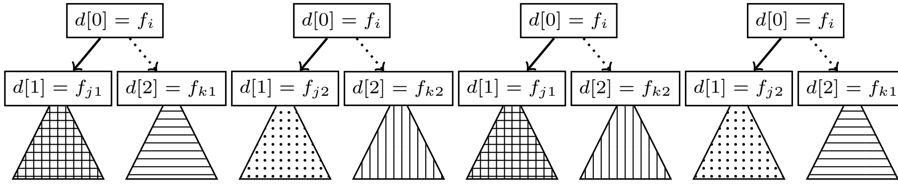
$$d[i] \neq 0 \Rightarrow (c[\text{left}(i)] \geq N_{\min} \wedge c[\text{right}(i)] \geq N_{\min}) \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (28)$$

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

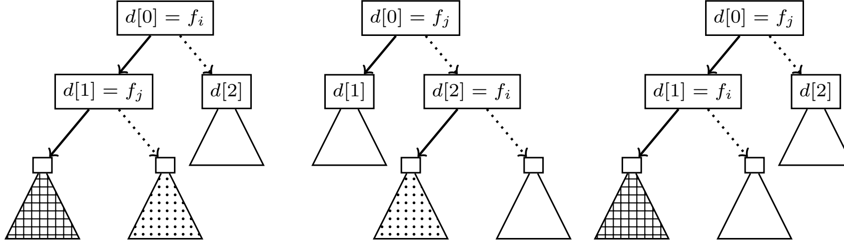
Adding a constraint CoverSizeSR for all of the nodes in the tree allows the computation of the exact values of the counters earlier in the tree and therefore helps prune earlier some candidate solutions. However adding them to all the decision nodes is not necessary. Constraints (12) and (13) can be 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).

5.2 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 3 details the pseudo-code of our algorithm.



(a) Independence of subtrees



(b) Redundant subtrees; identically shaded subtrees are identical

Fig. 4: Decompositions

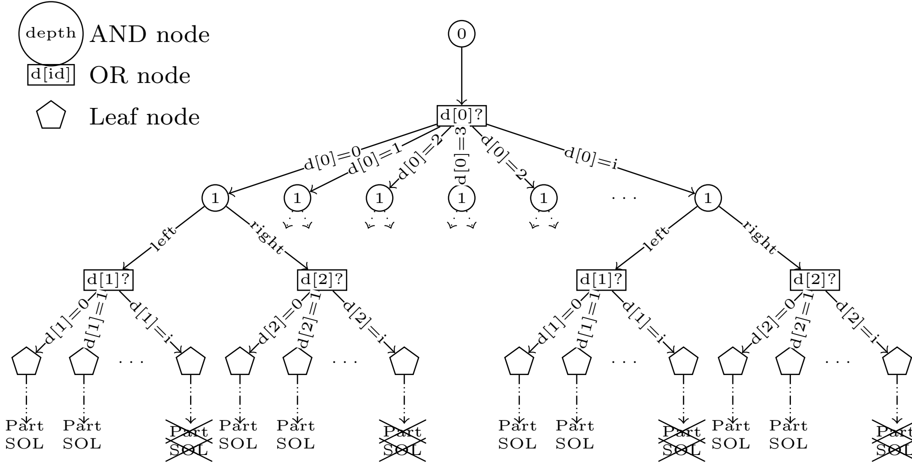


Fig. 5: AND/OR formulation of the search tree

5.2.1 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.

5.2.2 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

Algorithm 3: AND/OR formulation with cache and minimum pruning

```
1  Function search(currProblem: ∈ N^D):(Tree,Cost)
2      return ORnode(currProblem, ∞)
3  Function 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             trail.pushState()
13             try
14                 dom(d[currProblem]) ← {f}
15                 dom(e[currProblem]) ← {v|v < cost_best ∧ v ∈ dom(e[currProblem])}
                     // pruning by minimisation
16                 if currProblem ∈ N^N then
17                     (sol_tree, cost_tree) ← ANDnode(currProblem, cost_best, f)
18                 else
19                     sol_tree ← Tree(featureID : f left : null right : null)
20                     cost_tree ← e[currProblem].value
21                 if cost_best > cost_tree then
22                     cost_best ← cost_tree
23                     sol_best ← sol_tree
24             catch Inconsistency
                                                        // node have failed
25             trail.restoreState()
26         storage.add(prefix_hash, (sol_best, cost_best))       // new sol cached
27         return (sol_best, cost_best)
28 Function ANDnode(currProblem: ∈ N^D, cost_ub, f_root):(Tree,Cost)
29     (sol_left, cost_left) ← ORnode(left(currProblem), cost_ub)          // 1st
30     if cost_left > cost_ub then
31         return (null, ∞)                    // pruning based on cost
32     (sol_right, cost_right) ← ORnode(right(currProblem), cost_ub − cost_left)   // 2nd
33     sol_tree ← Tree(featureID : f_root left : sol_left right : sol_right)
34     return (sol_tree, cost_left + cost_right)
```

subtree. This is illustrated at Fig. 4a. 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, 14] framework is well suited for this kind of decomposable problem. The search is composed of two types of search nodes: the OR nodes (line 3) and the AND nodes (line 28). An example of the search tree for a decision tree of depth 2 is shown at Fig. 5.

The AND node is responsible for computing the optimal value of the left child (line 29), then the right child (line 32), and finally returns the composed solution

(line 33). The OR node tests all the possible values for the root decision variable of `currProblem` (line 11). The static ordering used to select the next value to test follows the principle of entropy [7]. The entropy of a set of transaction S is computed using the number of transactions from each class, and is a well-known heuristic in standard algorithms for learning decision trees:

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

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:

$$\begin{aligned} Gain(f) = Entropy(D) &- \frac{|\{t \in D : D_{t,f} = 1\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 1\}) \\ &- \frac{|\{t \in D : D_{t,f} = 0\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 0\}). \end{aligned} \tag{30}$$

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 16). In the other case (i.e. if `currProblem` belongs to $\mathcal{N}^{\mathcal{E}}$), then we have already an optimal subtree (line 18). From all the values tested, the best sub-tree is kept (line 21) and returned (line 27).

5.2.3 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 4b 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 [16]. 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 26).

5.2.4 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 21) and the best cost found so far is set as upper bound of the error of the subtree (line 15). 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 30). 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 32). The search starts with an unbounded upper bound (∞). This is the default value if, prior to the search, no information is known about the optimal cost.

5.2.5 Implementation details

Oscar [18], the solver used in our experiment does not implement the AND/OR search tree framework. A simple AND/OR search can be easily implemented using a standard trail based solver [13]. The two main operations of a trailing system are the saveState() and the restoreState() methods. The first one is responsible for saving the current state of the solver and the second one to restore it. In a typical OR tree, a save is done before trying a new assignment and start a new OR node. The state is restored when the node is fully explored. In an AND/OR tree, the logic is the same. Algorithm 3 depicts, in the ORnode() method, where the save and restore are being done. Just before trying a new assignment, at line 12, a save of the state is performed. Then the assignment is tested and the node fully explored. The exploration of the node is embedded in an exception catching mechanism. In case of inconsistencies (i.e. a proof of no solutions) during the exploration of the node, an exception is thrown leading to the stop in the exploration. After the exploration, a restoration of the state is required (line 25). For further implementation details, the source code is available online[2].

6 Results

We compared our algorithm to two exact methods developed in earlier studies: BinOCT [26] and DL8 [16], both of which solve exactly the same optimization problem as our method. Notice that the optimal solution trees found by these algorithms are in most cases equal, therefore leading to the same model. Only in rare cases, two different trees are equally good. In any case, this can't be used as a criterion to efficiently differentiate the prediction efficiently between the exact

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

methods since they are both able to output each an optimal tree. Both studies have already evaluated the quality of the resulting trees experimentally. It was shown in [3] that the more optimal is a tree on the training set, the more accurate it is on a test set. These results were confirmed in [26,16]. Therefore 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.

6.1 First benchmark

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
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

The benchmark[3] is composed of instances from the CP4IM[4] and UCI [5] 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 [22] is time consuming, but it allows improving a solution when the search stagnates. DL8 is a dynamic programming approach. It computes a subset of the frequent

[3] Available in the repository
[4] `https://dtai.cs.kuleuven.be/CP4IM/datasets/`
[5] `https://archive.ics.uci.edu/ml/index.php`

Dataset	Depth	$N_{\min} = 1$ DL8 obj	DL8 t	BinOCT obj	BinOCT t	CP obj	CP t
anneal	2	**137***	1	**137***	206	**137***	**<1**
anneal	3	**112***	37	**112**	TO	**112***	**2**
anneal	4	∞	TO	121	TO	**91***	**142**
anneal	5	∞	TO	120	TO	**84**	TO
audiology	2	**10***	**<1**	**10***	60	**10***	**<1**
audiology	3	**5***	62	7	TO	**5***	**5**
audiology	4	∞	TO	**1**	TO	**1**	TO
audiology	5	∞	TO	4	TO	**0***	**3**
australian-credit	2	**87***	2	**87***	206	**87***	**<1**
australian-credit	3	**73***	124	86	TO	**73***	**9**
australian-credit	4	∞	TO	85	TO	**57**	TO
breast-wisconsin	2	**22***	2	**22***	44	**22***	**<1**
breast-wisconsin	3	**15***	103	16	TO	**15***	**6**
breast-wisconsin	4	∞	TO	15	TO	**7***	**493**
diabetes	2	**177***	1	180	TO	**177***	**<1**
diabetes	3	**162***	93	171	TO	**162***	**8**
diabetes	4	∞	TO	169	TO	**137**	TO
german-credit	2	**267***	2	**267**	TO	**267***	**<1**
german-credit	3	**236***	129	249	TO	**236***	**8**
german-credit	4	∞	TO	244	TO	**204**	TO
heart-cleveland	2	**60***	**<1**	**60***	312	**60***	**<1**
heart-cleveland	3	**41***	17	43	TO	**41***	**4**
heart-cleveland	4	**25***	515	39	TO	**25***	**265**
heart-cleveland	5	∞	TO	34	TO	**9**	TO
hepatitis	2	**16***	**<1**	**16***	8	**16***	**<1**
hepatitis	3	**10***	4	12	TO	**10***	**1**
hepatitis	4	**3***	54	10	TO	**3***	**49**
hepatitis	5	∞	TO	7	TO	**0***	**8**
hypothyroid	2	**70***	4	**70***	178	**70***	**<1**
hypothyroid	3	**61***	122	62	TO	**61***	**4**
hypothyroid	4	∞	TO	62	TO	**53***	**183**
ionosphere	2	**32***	50	**32**	TO	**32***	**1**
ionosphere	3	∞	TO	29	TO	**22***	**328**
ionosphere	4	∞	TO	26	TO	**13**	TO
kr-vs-kp	2	**418***	2	**418**	TO	**418***	**<1**
kr-vs-kp	3	**198***	74	301	TO	**198***	**2**
kr-vs-kp	4	∞	TO	877	TO	**144***	**107**
kr-vs-kp	5	∞	TO	675	TO	**81**	TO
letter	2	∞	TO	813	TO	**599***	**1**
letter	3	∞	TO	813	TO	**369***	**108**
letter	4	∞	TO	∞	TO	**294**	TO

Table 2: Results (part 1) Time out (TO) = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

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 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 4 and Table 5 shows the results for two methods (DL8 and ours) and some variations of our approach (without the caching, labelled CP-c, and without the pruning

Dataset	Depth	$N_{\min} = 1$					
		DL8		BinOCT		CP	
		obj	t	obj	t	obj	t
lymph	2	**22***	**< 1**	**22***	17	**22***	**< 1**
lymph	3	**12***	2	13	TO	**12***	**1**
lymph	4	**3***	43	8	TO	**3***	**42**
lymph	5	∞	TO	8	TO	**0***	**1**
mushroom	2	**252***	27	520	TO	**252***	**< 1**
mushroom	3	∞	TO	396	TO	**8***	**4**
mushroom	4	∞	TO	160	TO	**0***	**< 1**
pendigits	2	∞	TO	**153**	TO	**153***	**1**
pendigits	3	∞	TO	496	TO	**47***	**50**
pendigits	4	∞	TO	780	TO	**14**	TO
primary-tumor	2	**58***	**< 1**	**58***	5	**58***	**< 1**
primary-tumor	3	**46***	**< 1**	49	TO	**46***	**< 1**
primary-tumor	4	**34***	**2**	39	TO	**34***	4
primary-tumor	5	**26***	**14**	37	TO	**26***	71
segment	2	**9***	49	**9**	TO	**9***	**< 1**
segment	3	∞	TO	6	TO	**0***	**2**
segment	4	∞	TO	21	TO	**0***	**1**
soybean	2	**55***	**< 1**	**55***	19	**55***	**< 1**
soybean	3	**29***	2	42	TO	**29***	**1**
soybean	4	**14***	33	16	TO	**14***	**14**
soybean	5	**8***	**315**	24	TO	**8***	497
splice-1	2	**508***	143	522	TO	**508***	**< 1**
splice-1	3	∞	TO	574	TO	**224***	**125**
splice-1	4	∞	TO	1087	TO	**141**	TO
tic-tac-toe	2	**282***	**< 1**	**282***	10	**282***	**< 1**
tic-tac-toe	3	**216***	**< 1**	231	TO	**216***	**< 1**
tic-tac-toe	4	**137***	**3**	169	TO	**137***	**3**
tic-tac-toe	5	**63***	**16**	128	TO	**63***	64
vehicle	2	**75***	23	**75**	TO	**75***	**< 1**
vehicle	3	∞	TO	60	TO	**26***	**45**
vehicle	4	∞	TO	84	TO	**13**	TO
vote	2	**17***	**< 1**	**17***	8	**17***	**< 1**
vote	3	**12***	2	13	TO	**12***	**1**
vote	4	**5***	23	11	TO	**5***	**16**
vote	5	**1***	**248**	5	TO	**1***	394
yeast	2	**437***	2	**437**	TO	**437***	**< 1**
yeast	3	**403***	74	430	TO	**403***	**6**
yeast	4	∞	TO	412	TO	**366***	**287**
zoo-1	2	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**

Table 3: Results (part 2) Time out = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

using bounds, labelled CP-m) 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 6 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.

Dataset	Depth	$N_{\min} = 5$							
		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t
anneal	2	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**
anneal	3	**112***	31	**112***	**3**	**112***	**3**	**112***	4
anneal	4	**94***	591	**94***	**172**	**94***	257	**94***	296
anneal	5	∞	TO	**92**	TO	**92**	TO	**92**	TO
audiology	2	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**
audiology	3	**7***	2	**7***	**1**	**7***	**1**	**7***	2
audiology	4	**4***	**43**	**4***	56	**4***	55	**4***	74
audiology	5	**1***	512	**1***	534	**1***	**475**	**1**	TO
australian-credit	2	**87***	2	**87***	**< 1**	**87***	**< 1**	**87***	**< 1**
australian-credit	3	**74***	90	**74***	**11**	**74***	12	**74***	14
australian-credit	4	∞	TO	**60**	TO	66	TO	66	TO
breast-wisconsin	2	**22***	3	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**
breast-wisconsin	3	**15***	80	**15***	**8**	**15***	**8**	**15***	12
breast-wisconsin	4	∞	TO	**9**	TO	**9**	TO	**9**	TO
diabetes	2	**177***	1	**177***	**< 1**	**177***	**< 1**	**177***	**< 1**
diabetes	3	**162***	90	**162***	**10**	**162***	11	**162***	13
diabetes	4	∞	TO	**138**	TO	**138**	TO	**138**	TO
german-credit	2	**267***	2	**267***	**< 1**	**267***	**< 1**	**267***	**< 1**
german-credit	3	**236***	122	**236***	**11**	**236***	13	**236***	14
german-credit	4	∞	TO	**205**	TO	**205**	TO	**205**	TO
heart-cleveland	2	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**
heart-cleveland	3	**41***	15	**41***	**5**	**41***	8	**41***	7
heart-cleveland	4	**27***	404	**27***	**333**	**27***	528	**27***	595
heart-cleveland	5	∞	TO	**17**	TO	**17**	TO	18	TO
hepatitis	2	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**
hepatitis	3	**11***	2	**11***	**1**	**11***	2	**11***	2
hepatitis	4	**8***	**36**	**8***	62	**8***	86	**8***	99
hepatitis	5	**5***	**299**	6	TO	6	TO	8	TO
hypothyroid	2	**70***	3	**70***	**< 1**	**70***	**< 1**	**70***	**< 1**
hypothyroid	3	**62***	95	**62***	**4**	**62***	**4**	**62***	8
hypothyroid	4	∞	TO	**54***	**236**	**54***	323	**54***	570
ionosphere	2	**32***	48	**32***	**1**	**32***	**1**	**32***	**1**
ionosphere	3	∞	TO	**22***	**389**	**22***	443	**22**	TO
ionosphere	4	∞	TO	**16**	TO	**16**	TO	**16**	TO
kr-vs-kp	2	**418***	2	**418***	**< 1**	**418***	**< 1**	**418***	**< 1**
kr-vs-kp	3	**198***	63	**198***	**4**	**198***	**4**	**198***	7
kr-vs-kp	4	∞	TO	**144***	**214**	**144***	256	**144***	483
kr-vs-kp	5	∞	TO	**98**	TO	**98**	TO	132	TO
letter	2	∞	TO	**599***	**5**	**599***	**5**	**599***	**5**
letter	3	∞	TO	**369**	TO	**369**	TO	531	TO
letter	4	∞	TO	**296**	TO	**296**	TO	301	TO

Table 4: Results (part 1) Time out (TO) = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

Our method outperforms the two others on most of the instances. It could find and prove optimality on roughly 83% 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 [23].

Dataset	Depth	$N_{\min} = 5$ DL8 obj	DL8 t	CP obj	CP t	CP-c obj	CP-c t	CP-m obj	CP-m t
lymph	2	**22***	< **1**	**22***	< **1**	**22***	< **1**	**22***	< **1**
lymph	3	**13***	**1**	**13***	**1**	**13***	2	**13***	2
lymph	4	**7***	**15**	**7***	34	**7***	40	**7***	59
lymph	5	**4***	**166**	**4***	595	**4**	TO	**4**	TO
mushroom	2	**252***	24	**252***	< **1**	**252***	< **1**	**252***	< **1**
mushroom	3	∞	TO	**8***	**15**	**8***	**15**	**8***	44
mushroom	4	∞	TO	**0***	< **1**	**0***	< **1**	**0**	TO
pendigits	2	∞	TO	**153***	**2**	**153***	**2**	**153***	**2**
pendigits	3	∞	TO	**47***	**256**	**47***	268	**47***	415
pendigits	4	∞	TO	**15**	TO	19	TO	19	TO
primary-tumor	2	**58***	< **1**	**58***	< **1**	**58***	< **1**	**58***	< **1**
primary-tumor	3	**46***	< **1**	**46***	< **1**	**46***	< **1**	**46***	< **1**
primary-tumor	4	**40***	**1**	**40***	4	**40***	6	**40***	5
primary-tumor	5	**34***	**8**	**34***	65	**34***	162	**34***	104
segment	2	**9***	41	**9***	1	**9***	< **1**	**9***	1
segment	3	∞	TO	**2***	**69**	**2***	71	**2***	136
segment	4	∞	TO	**0***	186	**0***	**181**	**0**	TO
soybean	2	**55***	< **1**	**55***	< **1**	**55***	< **1**	**55***	< **1**
soybean	3	**29***	2	**29***	**1**	**29***	**1**	**29***	**1**
soybean	4	**15***	27	**15***	**21**	**15***	26	**15***	35
soybean	5	**13***	**239**	**13**	TO	**13**	TO	**13**	TO
splice-1	2	**508***	89	**508***	**1**	**508***	**1**	**508***	**1**
splice-1	3	∞	TO	**225***	**156**	**225***	188	**225***	249
splice-1	4	∞	TO	**142**	TO	**142**	TO	**142**	TO
tic-tac-toe	2	**282***	< **1**	**282***	< **1**	**282***	< **1**	**282***	< **1**
tic-tac-toe	3	**216***	< **1**	**216***	< **1**	**216***	< **1**	**216***	< **1**
tic-tac-toe	4	**137***	**3**	**137***	7	**137***	9	**137***	5
tic-tac-toe	5	**63***	**16**	**63***	83	**63***	282	**63***	167
vehicle	2	**75***	20	**75***	< **1**	**75***	< **1**	**75***	1
vehicle	3	∞	TO	**28***	**83**	**28***	85	**28***	143
vehicle	4	∞	TO	**17**	TO	**17**	TO	**17**	TO
vote	2	**18***	< **1**	**18***	< **1**	**18***	< **1**	**18***	< **1**
vote	3	**13***	**1**	**13***	**1**	**13***	**1**	**13***	**1**
vote	4	**6***	**13**	**6***	17	**6***	20	**6***	41
vote	5	**3***	**118**	**3***	234	**3***	300	4	TO
yeast	2	**437***	2	**437***	< **1**	**437***	< **1**	**437***	< **1**
yeast	3	**403***	70	**403***	**7**	**403***	9	**403***	**7**
yeast	4	∞	TO	**367***	**421**	**367**	TO	**367***	541
zoo-1	2	**0***	< **1**	**0***	< **1**	**0***	< **1**	**0***	< **1**

Table 5: Results (part 2) Time out = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (*t*, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

	$N_{\min} = 1$ DL8	BinOCT	CP	$N_{\min} = 5$ DL8	CP	CP-c	CP-m
Proven optimality	49(61%)	13(16%)	**68**(85%)	54(67%)	**65**(81%)	63(79%)	59(74%)
Best solution found	49(61%)	21(26%)	**80**(100%)	54(67%)	**79**(99%)	77(96%)	72(90%)
Fastest	17(21%)	1(1%)	**63**(79%)	26(32%)	**52**(65%)	36(45%)	27(34%)
Time out	31(39%)	67(84%)	**12**(15%)	25(31%)	**15**(19%)	17(21%)	21(26%)

Table 6: Summary of the results

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 *tic-tac-toe* benchmark with a depth of 5 improves its timing by 70% when adding the cache). The effect of the pruning is significant in some cases. On some benchmarks such as *mushroom*, *hypothyroid*, *ionosphere* or *vehicle*, the pruning improves greatly the solution (ex. on *hypothyroid* depth 4, the time is divided by 2.4). This improvement indicates that when searching for the best subtree, the best one is found early, prunning a fair amount of the search space concerning the subtree.

6.2 Second benchmark

Dataset	n	n^+	n^-	m
dexter	300	150	150	25736
dorothea	800	78	722	100000
dota2	92650	48782	43868	226

Table 7: Description of the instances (after binarization)

Dataset	Depth	$N_{\min} = 1$			
		DL8		CP	
		obj	t	obj	t
dexter	2	∞	TO	**135**	TO
dexter	3	∞	TO	**129**	TO
dexter	4	∞	TO	**124**	TO
dexter	5	∞	TO	**120**	TO
dorothea	2	∞	TO	**39**	TO
dorothea	3	∞	TO	**78**	TO
dota2	2	∞	TO	**42610***	33
dota2	3	∞	TO	**42107**	TO

Table 8: Results (part 3) Time out = 20 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

To evaluate the scaling of our method, we tested it on some bigger instances from the UCI website. The description of these instances is available in Tab. 7. The results with $N_{\min} = 1$ are available in Tab. 8 and with $N_{\min} = 5$ Tab. 8. A timeout of 20 minutes was used for these instances. These datasets required binarization first[6].

[6] Binarized versions available in the repository

Dataset	Depth	DL8 obj	DL8 t	CP obj	CP t	CP-c obj	CP-c t	CP-m obj	CP-m t
		$N_{\min} = 5$							
dexter	2	**136***	13	**136***	12	**136***	12	**136***	15
dexter	3	**130***	99	**130***	264	**130***	260	**130***	198
dexter	4	**125***	100	**125***	849	**125***	829	**125***	602
dexter	5	**120***	1066	**120**	TO	**120**	TO	**120**	TO
dorothea	2	∞	TO	**39**	TO	**39**	TO	**39**	TO
dorothea	3	∞	TO	**78**	TO	**78**	TO	**78**	TO
dota2	2	∞	TO	**42610***	97	**42610***	100	**42610***	124
dota2	3	∞	TO	**42107**	TO	**42107**	TO	**42107**	TO

Table 9: Results (part 3) Time out = 20 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

As shown by the *dota2* instance, in the line of the *letter* instance of the first benchmark, increasing the number of transactions affects less our algorithm than DL8. This is due to the use of the bitsets inside the CoverSize constraint. The *dexter* instance allows us to see that good performance can be achieved with a big number of features. Unfortunately, with too many features, as shown on the *dorothea* instance, even a depth-2 tree is not obtainaible within time out.

7 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 is 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.

Remark

This paper is an extended and improved version of the paper initially accepted to CP2019 in the journal fast track. The initial work was also presented in a 2-page summary abstract [24] at BNAIC2019, a national conference, and as a 4-page summary abstract [25], in the sister conference track at IJCAI20.

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: I.P. Gent (ed.) Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings, *Lecture Notes in Computer Science*, vol. 5732, pp. 173–187. Springer (2009). DOI 10.1007/978-3-642-04244-7_16. URL `https://doi.org/10.1007/978-3-642-04244-7_16`
5. Bonfietti, A., Lombardi, M., Milano, M.: Embedding decision trees and random forests in constraint programming. In: L. Michel (ed.) Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9075, pp. 74–90. Springer (2015). DOI 10.1007/978-3-319-18008-3_6. URL `https://doi.org/10.1007/978-3-319-18008-3_6`
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. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: International Conference on Principles and Practice of Constraint Programming, pp. 207–223. Springer (2016)
11. Hancock, T., Jiang, T., Li, M., Tromp, J.: Lower bounds on learning decision lists and trees. Information and Computation **126**(2), 114–122 (1996)
12. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is np-complete. Inf. Process. Lett. **5**(1), 15–17 (1976). DOI 10.1016/0020-0190(76)90095-8. URL `https://doi.org/10.1016/0020-0190(76)90095-8`
13. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck: MiniCP: A lightweight solver for constraint programming (2018). Available from `https://minicp.bitbucket.io`
14. 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)
15. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J.: Learning optimal decision trees with SAT. In: J. Lang (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, pp. 1362–1368. ijcai.org (2018). DOI 10.24963/ijcai.2018/189. URL `https://doi.org/10.24963/ijcai.2018/189`
16. 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)
17. Nijssen, S., Fromont, É.: Optimal constraint-based decision tree induction from itemset lattices. Data Min. Knowl. Discov. **21**(1), 9–51 (2010)
18. OscaR Team: OscaR: Scala in OR (2012). Available from `https://bitbucket.org/oscarlib/oscar`
19. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986). DOI 10.1023/A:1022643204877. URL `https://doi.org/10.1023/A:1022643204877`
20. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
21. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
22. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
23. 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)

24. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C., Schaus, P.: Learning optimal decision trees using constraint programming. In: K. Beuls, B. Bogaerts, G. Bontempi, P. Geurts, N. Harley, B. Lebichot, T. Lenaerts, G. Louppe, P.V. Eecke (eds.) Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019, *CEUR Workshop Proceedings*, vol. 2491. CEUR-WS.org (2019). URL `http://ceur-ws.org/Vol-2491/abstract109.pdf`
25. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C., Schaus, P.: Learning optimal decision trees using constraint programming (extended abstract). In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, Tokyo, Japan, 2020 (2020)
26. Verwer, S., Zhang, Y.: Learning optimal classification trees using a binary linear program formulation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 1625–1632. AAAI Press (2019). DOI 10.1609/aaai.v33i01.33011624. URL `https://doi.org/10.1609/aaai.v33i01.33011624`