Minimising Decision Tree Size as Combinatorial Optimisation*

Christian Bessiere[1], Emmanuel Hebrard[2], and Barry O'Sullivan[2]

[1] LIRMM, Montpelier, France
bessiere@lirmm.fr
[2] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{e.hebrard,b.osullivan}@4c.ucc.ie

Abstract. Decision tree induction techniques attempt to find small trees that fit a training set of data. This preference for smaller trees, which provides a learning bias, is often justified as being consistent with the principle of Occam's Razor. Informally, this principle states that one should prefer the simpler hypothesis. In this paper we take this principle to the extreme. Specifically, we formulate decision tree induction as a combinatorial optimisation problem in which the objective is to minimise the number of nodes in the tree. We study alternative formulations based on satisfiability, constraint programming, and hybrids with integer linear programming. We empirically compare our approaches against standard induction algorithms, showing that the decision trees we obtain can sometimes be less than half the size of those found by other greedy methods. Furthermore, our decision trees are competitive in terms of accuracy on a variety of well-known benchmarks, often being the most accurate. Even when post-pruning of greedy trees is used, our constraint-based approach is never dominated by any of the existing techniques.

1 Introduction

Decision trees [5] are amongst the most commonly used classifiers in real-world machine learning applications. Part of the attraction of using a decision tree is that it is easy to use and interpret. For example, consider the data set for a simple classification task in Figure 1(a). Each training example is defined by a set of weather features (outlook, temperature, humidity, windy) and a class: $+$ $(-)$ meaning I am happy (unhappy) to play outdoors under the given weather conditions. A decision tree for this training set is presented in Figure 1(b). The decision tree makes classifications by sorting the features of an instance through the tree from the root to some leaf node. At each internal node a test on a feature is performed, and each subtree corresponds to a possible outcome for that test. Classifications are made at the leaf nodes.

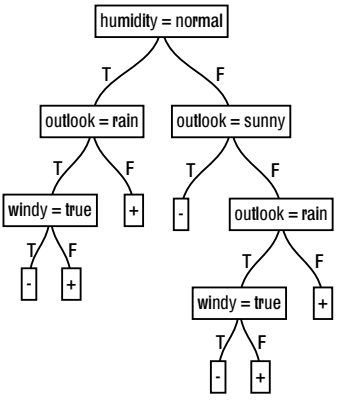
Traditional decision tree induction techniques attempt to find small trees that fit a training set of data. This preference for smaller trees is often justified as being consistent

* Bessiere is supported by the project CANAR (ANR-06-BLAN-0383-02). Hebrard and O'Sullivan are supported by Science Foundation Ireland (Grant number 05/IN/I886).

I.P. Gent (Ed.): CP 2009, LNCS 5732, pp. 173–187, 2009.

outlook	temp.	humidity	windy	play?
sunny	hot	high	false	-
sunny	hot	high	true	-
dull	hot	high	false	+
rain	mild	high	false	+
rain	cool	normal	false	+
rain	cool	normal	true	-
dull	cool	normal	true	+
sunny	mild	high	false	-
sunny	cool	normal	false	+
rain	mild	normal	false	+
sunny	mild	normal	true	+
dull	mild	high	true	+
dull	hot	normal	false	+
rain	mild	high	true	-

(a) An example data-set.



(b) A decision tree.

Fig. 1. An example decision tree learning problem

with the principle of Occam's Razor. Informally, this principle states that one should prefer simpler hypotheses. However, the majority of existing decision tree algorithms are greedy and rely on heuristics to find a small tree without search. While the decision trees found are usually small, there is no guarantee that much smaller, and possibly more accurate, decision trees exist. Finding small decision trees is often of great importance. Consider a medical diagnosis task in which each test required to diagnose a disease is intrusive or potentially risky to the well-being of the patient. In such an application minimising the number of such tests is of considerable benefit.

In this paper we study the problem of minimising decision tree size by regarding the learning task as a combinatorial optimisation problem in which the objective is to minimise the number of nodes in the tree. We refer to this as the *Smallest Decision Tree Problem*. We study formulations based on satisfiability, constraint programming, and hybrids with integer linear programming. We empirically compare our approaches against standard induction algorithms, showing that the decision trees we obtain can sometimes be less than half the size of those found by other greedy methods. Furthermore, our trees are competitive in terms of accuracy on a variety of well-known benchmarks, often being the most accurate. Even when post-pruning of greedy trees is used, our constraint-based approach is never dominated by any of the existing techniques.

The remainder of the paper is organised as follows. In Section 2 we present the technical background and define the problem we solve in this paper. We present a formulation of the problem using satisfiability (Section 3), constraint programming (Section 4), and a hybrid of constraint programming and linear programming (Section 5). Our experimental results are presented in Section 6. Finally, we position our approach with respect to the existing literature in Section 7, and conclude in Section 8 highlighting some directions for future work.

2 Background

SAT and Constraint Programming. A *propositional satisfiability* (SAT) formula consists of a set of Boolean variables and a set of clauses, where a clause is a disjunction

of variables or their negation. The SAT problem is to find an assignment 0 (false) or 1 (true) to every variable, such that all clauses are satisfied. SAT is a very simple formalism, but it is extremely expressive, making the SAT problem NP-hard. In addition to its simplicity, SAT has the advantage that significant research effort has led to several extremely efficient SAT solvers being developed.

A *constraint network* is defined by a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for some subsets of variables. The constraint satisfaction problem is to find an assignment to each variable with a value from its domain such that all constraints are satisfied. *Constraint programming* (CP) involves expressing decision problems with constraint networks, called models of the problem to be solved.

The Smallest Decision Tree Problem. A standard approach to evaluating the quality of a machine learning technique is to first learn a hypothesis on a selected *training* set of examples, and then evaluate the *accuracy* of the learned hypothesis on a *test* set of examples. Although more sophisticated measurements can also be taken, we consider the problem of finding the smallest decision tree consistent with a training set of examples, which is known to be NP-Hard [2].

Let $\mathcal{E} = \{e_1, \ldots, e_m\}$ be a set of *examples*, that is, Boolean valuations of a set $\mathcal{F}$ of *features*, and let $\mathcal{E}^+$, $\mathcal{E}^-$ be a partition of $\mathcal{E}$. We denote by $e[f]$ the valuation (0 or 1) of the feature $f \in \mathcal{F}$ in example $e \in \mathcal{E}$. Let $T = (X, U, r)$ be a binary tree rooted by $r \in X$, where $L \subseteq X$ denotes the set of leaves of T. A *decision tree* based on T is a labelled tree in which each internal node $x \in X \setminus L$ is labelled with an element of $\mathcal{F}$, denoted by $f(x)$. Each edge $(x, y) \in U$ is labelled with a Boolean $g(x, y)$, where $g(x, y) = 0$ if y is the left child of x and $g(x, y) = 1$ if y is the right child of x. The *size* of the decision tree is the number of nodes of T. Given $l \in L$, $p(l)$ denotes the path in T from the root r to leaf l. To each example $e \in \mathcal{E}$, we can associate the unique leaf $l(e) \in L$ such that every edge (x, y) in $p(l(e))$ is such that $e[f(x)] = g(x, y)$. A decision tree *classifies* a set of examples $\mathcal{E}$ iff for every pair $e_i \in \mathcal{E}^+, e_j \in \mathcal{E}^-$ we have $l(e_i) \neq l(e_j)$. Given a set of examples $\mathcal{E}$, we want to find a decision tree that classifies $\mathcal{E}$ with a minimum number of nodes. Alternatively, we can minimise the longest branch.

In the rest of the paper, we assume that all features admit a Boolean valuation. Categorical features can be encoded numerically, and a non-binary feature $f \in [1, \ldots, s]$ can be represented by a set of binary features $f_1, \ldots, f_s$. These Boolean features correspond to *equality* splits, that is where $f_v = 1$ stands for $f = v$ and $f_v = 0$ stands for $f \neq v$. It is standard in machine learning to split numerical data with a *disequality* split ($f[e] \leq v$ *or* $f[e] > v$). In order to allow these two types of split, we add to each example a second set of Boolean features $f'_1, \ldots, f'_s$ standing for disequality splits. That is, where $f'_v = 1$ stands for $f \leq v$ and $f'_v = 0$ stands for $f > v$. For instance, let $f^1 \in [1..4], f^2 \in [1..4], f^3 \in [1..4]$ be three features and $e = \langle 2, 4, 1 \rangle$ be an example. The binary encoding would yield the following example: 0100 0001 1000 0111 0001 1111, on the set of Boolean features: $\{f^1_1, ..f^3_4, f'^1_1, ..f'^3_4\}$.

3 A SAT-Based Encoding

We first introduce a SAT model to find decision trees. This baseline approach requires a large number of clauses to represent the problem. Furthermore, guiding search with

the generic heuristics of SAT solvers is not efficient (see Section 6). However, this approach underlines the critical aspects of this problem that need to be addressed in order to develop an efficient approach.

Given a binary tree $T = (X, U, r)$ and a training set $\mathcal{E}$, we present a SAT formula that is satisfiable iff there is a decision tree based on T that classifies $\mathcal{E}$.

Intuition. Given a set of features $\mathcal{F} = \{a, b, q, r\}$, suppose there are two examples e_i in $\mathcal{E}^+$ and e_j in $\mathcal{E}^-$ that have a similar value on the set of features $eq(e_i, e_j) = \{a, b\}$ (with $e_i[a] = e_j[a] = 0$, $e_i[b] = e_j[b] = 1$) and that differ on the set of features $\mathcal{F} \setminus eq(e_i, e_j) = \{q, r\}$. The SAT encoding has to ensure that e_i and e_j are not associated with the same leaf. Examples e_i and e_j are not both associated with a given leaf $l \in L$ iff there exists an edge $(x, y) \in p(l)$ such that:

$$f(x) \in \mathcal{F} \setminus eq(e_i, e_j) \ \vee \ (f(x) \in eq(e_i, e_j) \ \& \ g(x, y) \neq e_i[f(x)]).$$

The first case ensures that if $l(e_i)$ and $l(e_j)$ have x as a common ancestor, they appear in one of the two subtrees rooted in x; the second case ensures that none of $l(e_i), l(e_j)$ is equal to l since they will both branch on the opposite child of x.

Encoding. For every node $x \in X \setminus L$, for every feature $f \in \mathcal{F}$, we introduce a literal t_{xf}, whose value 1 will mean that node x is labelled with feature f. For each pair $e_i \in \mathcal{E}^+$ and $e_j \in \mathcal{E}^-$, for each leaf $l \in L$, we build a clause that forbids e_i and e_j to be classified at l. On the example above, suppose there is a path $p(l) = (x_1, x_2, l)$ in the tree such that x_2 is the left child of x_1 and l is the right child of x_2. We would add the clause: $t_{x_1q} \vee t_{x_1r} \vee t_{x_2q} \vee t_{x_2r} \vee t_{x_1b} \vee t_{x_2a}$. t_{x_1q} means x_1 is labelled with a feature that discriminates between e_i and e_j because $q \in \mathcal{F} \setminus eq(e_i, e_j)$. t_{x_1b} means the feature labelling x_1 will classify both e_i and e_j in the branch that does not lead to l because $p(l)$ uses the left child of x_1 whereas $e_i[b] = e_j[b] = 1$. Formally, we build the clauses:

$$\begin{aligned} &\Big(\bigvee_{(x,y)\in p(l), f\in eq(e_i,e_j) \ | \ g(x,y)\neq e_i[f]} t_{xf} \Big) \\ &\vee \Big(\bigvee_{(x,y)\in p(l), f\in \mathcal{F}\setminus eq(e_i,e_j)} t_{xf} \Big) \\ &\forall (e_i, e_j) \in \mathcal{E}^+ \times \mathcal{E}^-, \forall l \in L. \end{aligned} \tag{1}$$

The following clauses ensure that each node is labelled with at most one feature:

$$(\neg t_{xf} \ \vee \ \neg t_{xf'}), \quad \forall x \in X \setminus L, \forall f, f' \in \mathcal{F}. \tag{2}$$

By construction, a solution to the SAT formula defined above completely characterises a decision tree. Let M be such a solution. A node $x \in X \setminus L$ will be labelled with $f \in \mathcal{F}$ iff $M[t_{xf}] = 1$.

We add redundant clauses specifying that two nodes on a same path should not take the same feature as it speeds up the resolution process:

$$\bigwedge_{(x,y)\in p(l),(x',y')\in p(l),x\neq x'} (\neg t_{xf} \ \vee \ \neg t_{x'f}), \quad \forall l \in L, \forall f \in \mathcal{F}. \tag{3}$$

Complexity. Given $n = |X|, k = |\mathcal{F}|, m = |\mathcal{E}|$, the number of literals is in $O(nk)$. Observe that the number of literals is independent of the size of $\mathcal{E}$. However, the number of clauses strongly depends on the size of $\mathcal{E}$. We build at most $m^2 \cdot n/2$ clauses of type (1), each of length in $O(kn)$, and $n/2 \cdot k^2$ clauses of type (2), each of length in $O(1)$, which gives a space complexity in $O(kn^2m^2 + nk^2)$. There are at most $n/2 \cdot k$ conjunctions of n^2 binary clauses of type (3), which gives an extra space in $O(kn^3)$. Observe that we approximate the depth of T by n. This is a brute force approximation. If the tree is balanced, the depth will be in $O(log(n))$.

Observation. Our encoding has an interesting characteristic: it deals with 'useless' nodes for free. A node $x \in X \setminus L$ is useless if none of the examples in $\mathcal{E}$ will go through x to be classified, that is, $\forall e \in \mathcal{E}, x \notin p(l(e))$. In our encoding a node x is useless if it is not assigned any feature, that is, $\forall f \in \mathcal{F}, t_{xf} = 0$ in the solution. We then can add an extra type of redundant clauses to avoid decision trees going through useless nodes before reaching a real node:

$$\bigvee_{f \in \mathcal{F}} (t_{xf}) \vee \neg t_{yf'}, \quad \forall (x, y) \in U, y \in X \setminus L, \forall f' \in \mathcal{F}. \tag{4}$$

There are nk such clauses, each of size in $O(k)$, which gives a total size in $O(k^2n)$.

4 A CP Model

The SAT encoding introduced in the previous section has several drawbacks. It does not scale well with the number of examples in the training set, and even less so with the depth of the decision tree. This latter problem is because the binary tree we encode is a 'superset' of the decision tree we find, and is fixed in advance. Moreover, when the number of examples m is large it would be too costly to maintain variables or clauses representing examples. We therefore introduce a special kind of set variable, where only the lower bound is stored and can be pruned. Usually, a set variable is represented using two reversible sets, one standing for the elements that *must* be in the set (lower bound) and one for the elements that *can* be in the set (upper bound). We implement these simplified set variables using a single reversible list of integers, representing the lower bound. The upper bound is implicit and the only possible operation is to shrink it to match the lower bound. This type of variable allows us to reason about large sets (sets of examples here), at a very low computational cost. Another observation from the SAT encoding is that starting from a complete tree is impractical, even for relatively small depths. We therefore use the expressivity of CP to get around this problem. We do not fix the binary tree on which to label. We simply assume an upper bound n on the number of nodes, and seek the smallest decision tree with n nodes or less. That is, both the tests to perform *and* the topology of the tree are decided within the model. We can therefore find potentially deep trees with a relatively small number of nodes.

4.1 Variables

We label nodes with integers from 1 to n, then for all $i \in [1..n]$ we introduce the following variables:

- $P_i \in [1..n]$: the index of the parent of node i.
- $L_i \in [1..n]$: the index of the left child of node i.
- $R_i \in [1..n]$: the index of the right child of node i.
- $N_i \in [0..2]$: the number of children of node i.
- $F_i \in [1..k]$: the index of the feature tested at node i.
- $D_{ij} \in \{0, 1\}$: "node j is a descendant of node i".
- $\emptyset \subseteq E_i \subseteq \{1, \ldots, m\}$: the subset of $\mathcal{E}$ such that the leaves associated with elements in E_i are all descendants of node i. We shall use the notation E_i^+ (resp. E_i^-) for $(E_i \cap \mathcal{E}^+)$ (resp. $(E_i \cap \mathcal{E}^-)$).
- $UB \in [0..n]$: an upper bound on the size of the decision tree (initialised to $n + 1$ and set to the size of the smallest decision tree found so far)

4.2 Constraint Program

The graph defined on nodes $\{1, \ldots, n\}$ with an edge (i, j) iff $P_j = i$ must form a tree. We use the TREE global constraint to enforce this requirement [4]. Notice that this constraint uses a data structure to store the set of descendants of every node. We make it explicit by using the Boolean variables D. We use a slightly modified version of the constraint that ensures the resulting graph is, in fact, composed of a single tree, and possibly a set of unconnected nodes; a node i is connected iff $P_i \neq i$ or $\exists j \neq i,\ P_j = i$. We, therefore, can add the constraint $\text{TREE}(P, D)$ to the model.

Next, we channel the variables N, L, R and P with the following constraints, thus making sure that the tree is binary.

$$\forall i \neq j \in [1..n],\ P_j = i \Leftrightarrow ((L_i = j) \text{ xor } (R_i = j)). \tag{5}$$

$$\forall i,\ N_i = \sum_{j \neq i} P_j = i. \tag{6}$$

The next constraint ensures that no feature is tested twice along a branch.

$$D_{ij} = 1 \Rightarrow F_i \neq F_j. \tag{7}$$

Now we introduce some constraints to ensure that for all i, the variables E_i stand for the set of examples that shall be tested on node i.

$$L_i = j \Rightarrow E_j = \{k \mid k \in E_i \ \wedge\ e_k[F_i] = 0\}. \tag{8}$$

$$R_i = j \Rightarrow E_j = \{k \mid k \in E_i \ \wedge\ e_k[F_i] = 1\}. \tag{9}$$

For each node i, we ensure that unless all examples are classified, (i.e., there is no pair of examples with opposite polarity agreeing on the feature tested on this node) it cannot be a leaf.

$$\exists k \in E_i^+ \ \wedge\ \exists k' \in E_i^- \ \wedge\ e_k[F_i] = e_{k'}[F_i] \Rightarrow N_i > 0. \tag{10}$$

4.3 Inference

We introduce a number of implied constraints to improve this model. The first constraint ensures that the feature tested at a given node splits the examples in a non-trivial way.

$$\exists k \in E_i^+, \exists k' \in E_i^-, \text{ s.t. } e_k[F_i] \neq e_{k'}[F_i]. \tag{11}$$

This constraint does not improve the search, but it does help the subsequent constraint to work effectively. When, for a given node, every feature splits the examples so that both positive and negative examples are represented left and right, we know that this node will need not one, but two children. Let E be a set of examples and f be a feature, we denote by $l(f, E)$ (resp. $r(f, E)$) the cardinality of the subset of E that will be routed left (resp. right) when testing f.

$$l(F_i, E_i^+) \cdot l(F_i, E_i^-) \cdot r(F_i, E_i^+) \cdot r(F_i, E_i^-) \neq 0 \Rightarrow \; N_i = 2. \tag{12}$$

Due to the previous constraint, we can compute a lower bound on the number of past and future nodes that will be required. A simple sum constraint ensures we do not seek trees larger than one already found:

$$\sum_{i \in [1..n]} N_i < UB. \tag{13}$$

4.4 Symmetry Breaking

A search algorithm over the constraint model might repeatedly explore isomorphic trees where only node labellings change, significantly degrading performance. To avoid this, we ensure that the trees are ordered from root to leaves and from left to right by adding the following constraints:

$$\forall i \in [1..n-1], \; P_i \leq min(i, P_{i+1}). \tag{14}$$

$$\forall i \in [1..n], \; i \leq R_i \leq 2 * i + 2. \tag{15}$$

$$\forall i \in [1..n], \; i \leq L_i \leq min(2 * i + 1, R_i). \tag{16}$$

4.5 Search

Even when adding implied constraints and symmetry breaking constraints, the problem is often too large for the model above to explore a significant part of the search space in a reasonable amount of time. Thus, it is critical to use an efficient search heuristic in order to find good solutions quickly. We used the well known *information gain* heuristic, used in standard decision tree learning algorithms, as a search strategy. Therefore, the first branch explored by our constraint model is similar to that explored by C4.5 [6].

We also observed that diversifying the choices made by the search heuristic (via randomization) and using a *restart strategy* was beneficial. We used the following method: instead of branching on the feature offering the best information gain, we randomly picked among the three best choices. This small amount of randomization allowed us to restart search after unsuccessful dives. Each successive dive is bounded by the number of fails, initialised to 100 and then geometrically incremented by a factor of 1.5.

5 Hybrid CP and LP Model

Next we discuss a promising inference method to deduce a good lower bound on the number of nodes required to classify a set of examples. Consider a partial solution of the CP model, such that when a node i of the decision tree is assigned (that is, the parent of i and the feature tested on i are both known) then its parent is also assigned. It follows that the set of examples tested on an assigned node is perfectly known. We can compute a lower bound on the number of nodes required to classify this set of examples. By summing all these lower bounds for every assigned node without assigned children, we obtain a lower bound on the number of extra nodes that will be necessary to classify all yet unclassified examples. If this lower bound is larger than the number of available nodes we can backtrack, cutting the current branch in the search tree.

Consider a pair of examples (e_i, e_j) such that $e_i \in \mathcal{E}^+$ and $e_j \in \mathcal{E}^-$. We define $\delta(e_i, e_j)$ to be the set of *discrepancies* between examples e_i and e_j as follows: $\delta(e_i, e_j) = \{f \mid e_i[f] \neq e_i[f]\}$. Furthermore, we denote by C the corresponding collection of sets: $C(\mathcal{E}) = \{\delta(e_i, e_j) \mid e_i \in \mathcal{E}^+ \wedge e_j \in \mathcal{E}^-\}$. A *hitting set* for a collection $S_1, \ldots, S_n$ of sets is a set H such that $H \cap S_i \neq \emptyset, i \in 1..n$.

Theorem 1. *If a decision tree classifies a set of examples $\mathcal{E}$, the set of features tested in the tree is a hitting set of $C(\mathcal{E})$.*

Proof. Let F_T be the set of features tested in the decision tree and let $e_i \in \mathcal{E}^+$ and $e_j \in \mathcal{E}^-$. Clearly, in order to classify e_i and e_j, at least one of the features for which e_i and e_j disagree must be tested. That is, we have $F_T \cap \delta(e_i, e_j) \neq \emptyset$. Hence F_T is a hitting set for $C(\mathcal{E})$. □

Consequently, the size of the minimum hitting set on $C(\mathcal{E})$ is a lower bound on the number of distinct tests, and hence of nodes of a decision tree for classifying a set of examples $\mathcal{E}$. At the root node, this hitting set problem might be much too hard to solve, and moreover, it might not be a tight lower bound since tests can be repeated several times on different branches. However, during search on the CP model described above, there shall be numerous subtrees, each corresponding to a subset of $\mathcal{E}$, for which solving, or approximating the hitting set problem might give us a valuable bound. We can solve the hitting set problem using the following linear program on the set of Boolean variables $V = \{v_f \mid f \in \mathcal{F}\}$:

$$\text{minimise} \sum_{f \in \mathcal{F}} v_f \text{ subject to : } \forall c \in C(\mathcal{E}), \sum_{f \in c} v_f \geq 1.$$

This linear program can be solved efficiently by any LP solver. At each node of the *search* tree explored by the CP optimiser, let OP be the set of nodes of the *decision* tree whose parent is known (assigned) but children are unknown (not yet assigned). For each node $i \in OP$, we know the exact set of examples E_i to be tested on i. Therefore, a lower bound $lb(i)$, computed with the LP above, of the cardinality of the associated MINIMUM HITTING SET problem is also a valid lower bound on the number of descendants of i. Let I be the set of nodes (of the *decision* tree) already assigned, and

UB be the size of the smallest tree found so far. We can replace Constraint 13 with the following constraint:

$$|I| + \sum_{i \in OP} lb(i) < UB. \tag{17}$$

In order to avoid computing large linear relaxations too often, we use a *threshold* on the cardinality of $C(E_i)$ that is $|E_i^+| \times |E_i^-|$. Whenever this cardinality is larger than the threshold, we use $N_i + 1$ instead of $lb(i)$ in Constraint 17.

6 Experimental Results

We performed a series of experiments comparing our approach against the state-of-the-art in machine learning, as well as studying the scalability and practicality of our optimisation-based methods. An important distinction between our approach and standard greedy decision tree induction algorithms, is that we seek the *smallest* tree that has *perfect* classification accuracy on the training set. In this sense, our approach can be regarded as a form of knowledge compilation in which we seek the smallest compiled representation of the training data [3]. Standard decision tree induction algorithms can be forced to generate a tree that also has perfect classification accuracy on the data, but these trees tend to be large, since they *overfit* the training data. To overcome this overfitting, greedy methods are often post-pruned by identifying sub-branches of the tree that can be removed without having too significant an impact on its accuracy.

In our experiments, therefore, we compared the decision trees obtained from our optimisation approach against the decision trees obtained from standard tree induction methods, both unpruned and pruned. The results clearly show that the constraint programming approach produces very accurate trees, that are smaller than those found using standard greedy unpruned methods, and in a scalable manner. Even when compared against pruned trees, the accuracy of our decision trees is never the worst, and is often competitive with, or exceeds that of pruned trees built using standard methods.

All our experiments were run on a 2.6GHz Octal Core Intel Xeon with 12Gb of RAM running Fedora core 9. In Table 1, we report some characteristics (number of examples and features) of the benchmarks used in our experiments.

6.1 The Scalability of the SAT Encoding

In Table 1 we give the space complexity, in bytes, of the CNF encoding for each data set, assuming a maximum depth of 4 for the decision tree. In most cases, however, the depth of minimal trees is much larger. Recall that the space complexity of the SAT

Table 1. Characteristics of the data-sets, and the sizes of the corresponding SAT formulae

Benchmark	Weather	Mouse	Cancer	Car	Income	Chess	Hand w.	Magic	Shuttle	Yeast
#examples	14	70	569	1728	30162	28056	20000	19020	43500	1484
#features	10	45	3318	21	494	40	205	1887	506	175
CNF size (depth 4)	27K	3.5M	92G	842M	354G*	180G	248G	967G*	118G*	13G

formula increases exponentially with depth. Therefore, the reformulation is too large in almost all of our data sets (the results marked with an asterisk (*) were obtained using only 10% of the examples in the data set).

In order to study the behaviour of a SAT solver on this problem we ran SAT4J[1] on the SAT encoding of the two smallest data sets (`Weather` and `Mouse`). We used the depth of the smallest tree found by the CP model (see Section 6.2) to build the SAT encoding. To minimise the size of the decision tree, SAT4J features an AtLeastK constraint ensuring that the number of 0's in a model is at least a given number K. The value of K is initialised to 0, and on each successful run, we set K to 1 plus the number of 0's in the previous model. We stop when either SAT4J returns false, or a time cutoff of 5 minutes has elapsed without improving the current model. We report the runtime for finding the best solution (sol.) and also the total elapsed time, including the time spent on proving or attempting to prove optimality (tot.). We also report the size (nodes) of the smallest tree found.

Benchmark	SAT model		
	time (sol.)	time (tot.)	tree size
`Weather`	0.14	0.37	9
`Mouse`	277.27	577.27	15

It is remarkable how well SAT4J can handle such large CNF files. For instance, the encoding of `Mouse` involves 74499 often very large clauses. However, it was clear the SAT method does not scale since these two tiny data sets produced large formulas. On the one hand, for the smaller data-set (`Weather`) SAT4J quickly found an optimal decision tree, but was slightly slower than the CP method (0.06s). On the other hand, for the larger data-set (`Mouse`), the CP model found a decision tree of 15 nodes in 0.05s, whilst the SAT required 277.27s to find a solution of equal quality but failed to prove its optimality within the 5 minute time limit.

6.2 The CP Model

In this experiment we compared the size of the decision trees produced by our CP classifier with respect to standard implementations of C4.5. We compare our results in terms of tree size and accuracy against Weka [8] and ITI [7], using a number of data-sets from the UCI Machine Learning Repository[2]. For each data set, and for a range of ratios ($\frac{|training\ set|}{|test\ set|}$) we produced 100 random training sets of the given ratio by randomly sampling the whole data-set, using the remainder of the data for testing. Each classifier is trained on the same random sample. We report averages over the 100 runs for each classifier.

We first compare the size of the decision trees when they are complete (100% accurate classification) on the training data. Therefore, we switched off all *post-pruning* capabilities of Weka and ITI. Moreover, Weka also *pre-prunes* the tree, that is, it does not expand subtrees when the information gain measure becomes too small.

[1] `http://www.sat4j.org`
[2] `http://archive.ics.uci.edu/ml/`

We, therefore, modified WEKA to avoid this behaviour[3]. The following command lines were used for WEKA and ITI, respectively: `java weka.classifiers.trees.J48 -t train_set -T test_set -U -M 0` and `iti dir -ltrain_set -qtest_set -t`. The CP optimiser was stopped after spending five minutes without improving the current solution. We report the size of the tree found in the first descent of the CP optimiser and the size of the smallest tree found. We also report some search information – number of backtracks and runtime in seconds – to find the smallest tree.

The results for these unpruned decision trees are reported in the columns 'C4.5, no pruning' and 'cp' of Table 2. One could imagine that the information gain heuristic would be sufficient to find near-minimal trees with ITI or WEKA. The results show that this is not the case. The decision trees computed by ITI or WEKA without pruning are far from being minimal. Indeed C4.5 does not actively aim at minimising the tree size. Smaller decision trees can be found, and our CP model is effective in doing so.

It is somewhat surprising that even the first solution of the CP model is often better (in terms of tree size) than that of WEKA or ITI. This can be explained by the fact that we turn the data set into a numerical form, and then systematically branch using either equality or disequality splits. On the other hand, WEKA uses only equality splits on categorical features, and disequality splits on the numerical features. Our method allows tests with better information gain in certain cases.

In the rightmost columns of Table 2, we report the size of the pruned decision trees computed by ITI and WEKA. When compared against the pruned C4.5 trees, the CP tree is always dominated in terms of size. However, two points should be noted about this. Firstly, we have made no attempt to post-prune the CP trees. If we did so, we could expect a reduction in tree size, possibly comparable to that obtained for the trees generated using C4.5. Secondly, after pruning, the decision tree is no longer guaranteed to have 100% classification accuracy on the original training set.

Table 3 presents a detailed comparison of classification accuracy between the trees built using our CP approach and those built using WEKA and ITI, both pruned and unpruned. For each of the standard approaches we present the average classification accuracy of its trees based on 100 tests. In addition, we present the complement to 1 of the p-value, obtained from a paired t-test performed using the statistical computing system R[4]; this statistic is presented in the column labelled 'sig' (for significance). Suppose that for two methods, their average accuracy x and y over 100 runs are such that $x < y$, this value $(1 - p)$ can be interpreted as the probability that x is indeed less than y. For each reported average accuracy, we compute the significance of its relation to the CP accuracy. We regard a difference as statistically significant if the corresponding 'sig' value is at least 0.95. For example in the first line of the table the unpruned WEKA accuracy is 91.54, while the CP accuracy is 91.66. The significance of this difference is 0.42 which means that this is not a statistically significant difference.

The two right-most columns of Table 3 indicate the relative performance of the CP model. We assume that method A gives better trees than method B iff the accuracy is significantly better, and we define a dominance relation with respect to the CP model,

[3] This change was made in consultation with the authors of WEKA to ensure the system was not adversely affected.

[4] `http://www.r-project.org/`

Table 2. A comparison of the sizes of decision trees obtained from WEKA without pruning (WEKA), WEKA with pruning (WEKA (p)), ITI without pruning (ITI), ITI with pruning (ITI (p)), and CP. We also present statistics on the running time of the CP approach.

Benchmark	Prop.	C4.5, no pruning		cp				C4.5, pruning	
		WEKA	ITI	first	best			WEKA (p)	ITI (p)
		size	size	size	size	time (s)	backtracks	size	size
Cancer	0.2	11.66	11.94	10.92	9.22	47.12	17363.04	7.76	5.00
	0.3	16.12	16.74	14.30	12.48	27.22	7411.31	10.26	7.28
	0.5	23.48	25.98	20.40	18.48	45.05	8448.93	15.08	10.72
	0.7	31.10	36.18	26.22	24.26	38.00	6361.05	20.56	12.76
	0.9	37.98	41.56	32.40	30.08	57.92	7801.57	23.46	15.08
Car	0.05	30.09	23.38	24.82	18.52	8.48	250851.11	12.30	10.92
	0.1	46.67	37.20	40.16	30.12	26.32	1228232.51	18.98	17.04
	0.2	70.97	55.26	59.82	47.70	41.60	1840811.13	29.04	27.40
	0.3	87.67	68.96	74.16	60.06	29.66	1107230.15	37.14	34.40
	0.5	114.51	84.50	93.32	75.60	33.52	1025980.81	52.66	47.56
	0.7	139.22	96.12	105.54	86.30	32.05	839389.68	60.70	57.76
	0.9	161.73	100.76	115.24	92.02	44.81	1086473.89	66.45	64.62
Income	0.01	185.86	108.82	85.12	76.22	35.99	141668.94	34.08	26.68
	0.015	265.23	160.89	123.60	112.87	36.81	108710.27	46.37	38.95
	0.05	791.03	534.69	390.65	364.83	63.99	56624.72	124.21	122.39
Chess	0.01	126.84	89.46	81.54	66.58	49.13	1486914.05	1.00	18.46
	0.015	172.36	130.52	119.60	98.90	48.86	1376762.87	1.06	29.64
	0.05	434.54	372.54	317.20	274.66	41.57	360814.88	1.00	108.92
	0.1	735.26	644.22	525.48	458.80	66.11	112665.88	34.08	217.88
Hand writing (A)	0.01	11.54	14.24	10.66	8.78	10.86	22525.52	5.66	5.98
	0.015	14.52	17.92	13.66	10.66	21.24	38645.64	5.66	7.56
	0.05	33.80	42.24	31.50	24.16	28.60	28678.29	10.36	10.78
	0.1	51.84	67.68	48.22	39.58	40.62	51219.42	17.54	18.28
	0.2	76.16	109.92	75.40	62.98	41.37	51613.47	29.60	34.94
	0.3	94.22	144.36	95.26	80.28	40.93	53276.16	43.36	46.52
Hand writing (B)	0.01	18.28	19.68	16.30	12.82	19.27	84162.49	5.96	5.02
	0.015	24.58	27.66	22.04	17.06	20.19	95053.05	9.24	7.00
	0.05	64.60	70.30	55.92	47.10	40.20	211587.58	23.42	17.22
	0.1	106.56	119.58	95.78	84.76	33.72	171899.21	36.92	33.04
	0.2	180.36	199.72	160.20	145.04	42.05	212308.28	72.16	58.04
	0.3	240.18	268.92	214.48	196.42	33.64	146700.04	105.56	78.68
Hand writing (C)	0.01	14.12	15.64	13.08	10.34	17.10	58892.16	7.04	4.70
	0.015	19.38	21.00	17.80	14.00	20.54	79994.81	8.82	6.10
	0.05	48.64	52.62	43.16	35.52	30.89	119385.76	18.40	16.64
	0.1	80.86	88.42	73.76	61.96	44.43	150533.20	29.74	30.72
	0.2	132.36	146.52	121.50	104.74	39.99	113072.51	49.72	52.52
	0.3	175.44	193.56	160.64	139.36	49.49	135286.41	66.56	72.24
Magic	0.01	52.74	57.24	48.44	44.02	39.19	38038.14	32.70	17.10
	0.015	76.14	82.42	69.76	64.28	32.59	22821.48	45.54	24.16
	0.05	234.70	257.02	204.38	195.94	95.97	20816.56	132.36	70.48
Shuttle	0.05	19.02	25.20	12.96	8.06	48.58	7107.41	10.38	13.14
	0.1	24.18	31.54	16.14	10.88	69.49	5639.98	15.22	17.56
	0.2	28.46	35.38	20.76	13.62	14.94	3691.08	18.04	24.98
Yeast CYT	0.05	33.26	38.70	30.26	25.30	37.16	353351.68	21.72	9.84
	0.1	67.50	76.42	59.08	50.88	39.86	291145.95	36.52	19.14
	0.2	130.30	147.70	113.36	103.78	37.57	215424.60	64.64	37.50
	0.3	197.26	222.16	172.22	157.12	28.72	136221.02	96.18	53.58
	0.5	324.30	364.62	284.10	263.42	41.36	167958.64	147.66	87.82
	0.7	450.06	507.30	395.16	371.46	35.35	100452.08	199.52	122.10
Yeast MIT	0.05	22.00	24.46	18.62	15.70	11.55	65659.79	8.68	5.10
	0.1	42.60	47.38	37.12	31.62	22.06	118185.26	13.36	9.04
	0.2	81.70	92.26	70.80	63.30	21.04	79821.79	22.66	19.40
	0.3	121.48	138.72	103.28	94.88	28.96	117579.25	29.86	27.64
	0.5	200.92	227.88	169.32	156.60	33.17	128659.70	42.00	42.00
	0.7	279.12	310.26	232.72	217.30	36.27	117305.35	59.10	57.76

Table 3. A comparison of the classification accuracies of decision trees obtained from WEKA without pruning (WEKA), WEKA with pruning (WEKA (p)), ITI without pruning (ITI), ITI with pruning (ITI (p)), and CP

Benchmark	Prop.	WEKA		WEKA (p)		ITI		ITI (p)		cp	Relation	
		accur.	sig.	accur.	sig.	accur.	sig.	accur.	sig.	accur.	versus all	versus complete
Cancer	0.2	91.54	0.42	91.71	0.17	91.09	0.97	90.86	0.99	91.66	*among best*	*among best*
	0.3	91.74	0.59	91.96	0.09	91.76	0.47	91.96	0.07	91.93	*among best*	*among best*
	0.5	92.57	0.86	92.95	0.23	92.43	0.96	92.90	0.07	92.88	*among best*	*among best*
	0.7	92.85	0.96	93.36	0.07	93.33	0.17	93.90	0.96	93.39	*among best*	*among best*
	0.9	93.22	0.83	93.50	0.51	93.66	0.24	93.78	0.01	93.78	*among best*	*among best*
Car	0.05	88.34	0.54	87.22	0.99	88.69	0.18	87.26	0.99	88.61	*among best*	*among best*
	0.1	91.07	0.99	89.41	1.00	92.24	0.92	90.58	0.99	91.84	*among best*	*among best*
	0.2	94.13	0.99	92.88	1.00	94.89	0.26	94.23	0.99	94.83	*among best*	*among best*
	0.3	95.45	1.00	94.05	1.00	96.25	0.89	95.47	1.00	96.44	*among best*	*among best*
	0.5	96.87	1.00	96.14	1.00	97.67	0.99	97.13	1.00	97.92	*best*	*best*
	0.7	97.60	1.00	96.97	1.00	98.49	0.99	97.93	1.00	98.72	*best*	*best*
	0.9	97.92	1.00	97.67	1.00	99.18	0.24	98.61	0.99	99.22	*among best*	*among best*
income	0.01	78.00	0.96	80.46	1.00	76.61	1.00	78.79	0.87	78.47	*incomp.*	*best*
	0.015	78.58	0.99	81.30	1.00	77.11	1.00	79.26	0.47	79.12	*incomp.*	*best*
	0.05	79.76	1.00	82.79	1.00	77.76	1.00	80.41	0.34	80.45	*incomp.*	*best*
Chess	0.01	84.43	0.99	89.94	1.00	84.15	0.99	87.75	1.00	85.16	*incomp.*	*best*
	0.015	85.43	0.99	89.93	1.00	84.62	1.00	87.50	1.00	86.35	*incomp.*	*best*
	0.05	89.11	1.00	89.94	1.00	87.59	1.00	88.23	1.00	90.40	*best*	*best*
	0.1	91.53	1.00	90.22	1.00	89.84	1.00	89.48	1.00	92.91	*best*	*best*
Hand writing (A)	0.01	97.48	0.60	98.04	0.99	96.84	0.98	97.14	0.69	97.33	*incomp.*	*among best*
	0.015	97.90	0.97	98.48	1.00	97.39	0.97	97.87	0.91	97.66	*incomp.*	*incomp.*
	0.05	98.48	0.20	98.87	1.00	98.26	0.99	98.63	0.99	98.47	*incomp.*	*among best*
	0.1	98.83	0.81	99.05	1.00	98.60	1.00	98.85	0.89	98.80	*incomp.*	*among best*
	0.2	99.17	0.99	99.23	1.00	98.91	1.00	99.04	0.99	99.10	*incomp.*	*incomp.*
	0.3	99.35	1.00	99.36	1.00	99.06	1.00	99.14	1.00	99.26	*incomp.*	*incomp.*
Hand writing (B)	0.01	94.65	0.41	95.79	1.00	94.79	0.17	95.98	1.00	94.75	*among worst*	*among best*
	0.015	95.03	0.97	96.01	0.99	95.12	0.87	96.17	1.00	95.33	*incomp.*	*among best*
	0.05	96.29	0.98	97.00	1.00	96.42	0.19	97.15	1.00	96.43	*incomp.*	*among best*
	0.1	96.99	0.22	97.42	1.00	96.95	0.49	97.35	1.00	96.98	*among worst*	*among best*
	0.2	97.62	0.99	97.80	1.00	97.51	0.09	97.74	1.00	97.51	*among worst*	*among worst*
	0.3	97.94	0.99	98.03	1.00	97.79	0.39	97.93	1.00	97.80	*among worst*	*among worst*
Hand writing (C)	0.01	95.81	0.65	96.14	0.99	95.84	0.72	96.33	0.99	95.67	*among worst*	*among best*
	0.015	96.15	0.89	96.61	0.97	96.27	0.49	96.49	0.77	96.35	*among worst*	*among best*
	0.05	97.38	0.86	97.75	1.00	97.41	0.72	97.57	0.94	97.47	*among worst*	*among best*
	0.1	97.99	0.25	98.22	1.00	97.90	0.97	98.02	0.72	97.98	*incomp.*	*among best*
	0.2	98.43	0.57	98.63	1.00	98.37	0.87	98.42	0.18	98.41	*among worst*	*among best*
	0.3	98.63	0.63	98.76	1.00	98.58	0.99	98.64	0.31	98.65	*incomp.*	*among best*
Magic	0.01	75.00	0.93	75.84	0.40	75.04	0.96	76.61	0.99	75.65	*incomp.*	*among best*
	0.015	76.32	0.89	77.35	0.99	76.08	0.98	77.84	0.99	76.69	*incomp.*	*best*
	0.05	78.29	1.00	79.66	0.99	78.21	1.00	80.50	1.00	78.98	*incomp.*	*best*
Shuttle	0.05	99.77	1.00	99.72	1.00	99.76	1.00	99.71	1.00	99.85	*best*	*best*
	0.1	99.85	1.00	99.80	1.00	99.85	1.00	99.79	1.00	99.91	*best*	*best*
	0.2	99.93	1.00	99.90	1.00	99.92	0.99	99.90	1.00	99.95	*best*	*best*
Yeast CYT	0.05	65.21	0.74	65.34	0.83	63.46	0.99	65.11	0.55	64.79	*among best*	*among best*
	0.1	66.53	0.86	67.19	0.99	64.72	0.99	66.87	0.98	66.14	*incomp.*	*among best*
	0.2	66.93	0.15	68.04	0.99	65.94	0.99	68.04	0.99	66.98	*incomp.*	*among best*
	0.3	67.87	0.98	69.19	1.00	66.42	0.99	68.84	1.00	67.28	*incomp.*	*incomp.*
	0.5	68.53	0.40	70.19	1.00	67.79	0.99	69.73	1.00	68.41	*incomp.*	*among best*
	0.7	69.09	0.94	70.99	1.00	68.75	0.43	70.39	1.00	68.59	*among worst*	*among best*
Yeast MIT	0.05	80.21	0.52	83.32	1.00	80.27	0.42	83.51	1.00	80.50	*among worst*	*among best*
	0.1	81.14	0.05	85.24	1.00	80.53	0.92	84.46	1.00	81.12	*among worst*	*among best*
	0.2	81.92	0.13	85.83	1.00	81.46	0.95	84.98	1.00	81.89	*incomp.*	*among best*
	0.3	82.30	0.31	86.34	1.00	81.89	0.89	85.76	1.00	82.22	*among worst*	*among best*
	0.5	82.62	0.85	86.77	1.00	82.48	0.98	85.99	1.00	82.90	*incomp.*	*among best*
	0.7	82.50	0.99	86.37	1.00	82.65	0.99	86.11	1.00	83.17	*incomp.*	*among best*

based on this pairwise relation. We say that the CP model is the best, denoted "*best*" (resp. the worst, denoted "*worst*") iff it gives trees that are significantly better (resp. worse) than all other methods. We say that the CP model is among the best, denoted "*among best*" (resp. among the worst, denoted "*among worst*") if there is no other method giving better (resp. worse) trees, and if it is not the best (resp. worst). Finally, we

Table 4. Runtime, #Backtracks & Tree size (CP vs CP+LP)

Benchmark	CP model			CP+LP model		
	time	bts	tree size	time	bts	size
	44.37	16206	9.24	**31.64**	**10421**	**9.20**
	27.61	7098	12.48	**21.48**	**4569**	**12.44**
Cancer	42.82	7910	**18.50**	**42.54**	**4461**	18.62
	8.75	250449	**18.52**	12.30	**7733**	**18.52**
	29.13	1390076	30.10	51.15	**33182**	**29.58**
Car	**48.21**	2206249	47.64	87.05	**32979**	**46.46**
	29.69	1106445	60.06	48.97	**15829**	**59.18**
	33.53	1025011	**75.60**	53.40	**17715**	75.64
	35.03	140917	**76.22**	59.98	**11875**	76.68
Income	**38.69**	114533	**112.81**	89.08	**11074**	113.26
	63.3	54219.82	364.83	119.14	6802.40	368.87

say that it is incomparable, denoted "*incomp.*", iff there exists at least one method giving better trees and one giving worse trees. We report this comparison for each instance. In the penultimate column ("versus all") we compare against all methods. The CP model is the best in 13% of the cases, among the best in 20% of the cases, incomparable in 45% of the cases, among the worst in 22% of the cases, and is never the worst. In the last column ("versus complete"), we compare against only complete methods, that is, WEKA and ITI without post-pruning. The CP model is the best in 25% of the cases, among the best in 64% of the cases, incomparable in 7% of the cases, among the worst in 4% of the cases, and is never the worst.

It is clear that the CP generated trees are almost always better than those generated by standard decision tree methods that do not do pruning. Also, when compared against methods that use post-pruning, the CP approach is not dominated by either one. This is an encouraging result since it suggests that while the CP generated trees are competitive with advanced decision tree induction methods, they can only be improved further if they were also post-pruned.

6.3 Improving the CP Model Using LP

The aim of this experiment was to assess if the linear relaxation method introduced in Section 5 can improve the CP model. We run it using the same setting as described in Section 6.2, and on 3 benchmarks and with an arbitrary threshold[5] of 600 that was a good compromise. In Table 4, we report the runtime and number of backtracks to find the best solution, as well as the quality (tree size) of this solution.

We observe that the search space explored by the CP+LP model can be orders of magnitude smaller (see #backtracks), but the runtime can still be slightly worse because of the overhead of solving the linear relaxation. Thus, even if it is difficult to judge if overall the method is better than the basic CP model, we can expect it to scale well on harder problems.

7 Related Work

Decision trees are usually constructed using greedy algorithms [5,6] relying on a search bias that attempts to find smaller trees. Finding the minimum sized tree is NP-Hard [2].

[5] As defined in Section 5.

[1] have proposed a technique for improving the accuracy of a decision tree by selecting the next attribute to test as the one with the smallest expected consistent sub-tree size estimated using a sampling technique. The use of combinatorial optimisation to improve decision trees has also been reported [9], where the focus has been on determining linear-combination splits for the decision tree. These papers are not concerned with minimising overall tree-size. Our work contrasts with these approaches since we take an extreme view of Occam's Razor and seek to find the minimum sized decision tree using alternative approaches from the field of combinatorial optimisation. We have found that size can usually be reduced considerably, without negatively impacting accuracy.

8 Conclusion

We have presented a variety of alternative approaches to minimising the number of nodes in a decision tree. In particular, we have shown that while this problem can be formulated as either a satisfiability problem or a constraint program, the latter is more scalable. Our empirical results show the value of minimising decision tree size. We find smaller trees that are often more accurate than, but never dominated by, those found using standard greedy induction algorithms.

References

1. Esmeir, S., Markovitch, S.: Anytime learning of decision trees. Journal of Machine Learning Research 8, 891–933 (2007)
2. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is NP-complete. Inf. Process. Lett. 5(1), 15–17 (1976)
3. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
4. Prosser, P., Unsworth, C.: Rooted tree and spanning tree constraints. In: Workshop on Modelling and Solving Problems with Constraints, held at ECAI 2006 (2006)
5. Quinlan, J.R.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986)
6. Quinlan, R.J.: C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
7. Utgoff, P.E., Berkman, N.C., Clouse, J.A.: Decision tree induction based on efficient tree restructuring. Machine Learning 29, 5–44 (1997)
8. Witten, I.H., Frank, E.: Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, San Francisco (2005)
9. Yüksektepe, F.Ü., Türkay, M.: A mixed-integer programming approach to multi-class data classification problem. European Journal of OR 173(3), 910–920 (2006)