ELSEVIER

Artificial Intelligence 83 (1996) 347–362

Artificial Intelligence

An efficient algorithm for optimal pruning of decision trees

Hussein Almuallim*

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, P.O. Box 801, Dhahran 31261, Saudi Arabia

Received September 1994

Abstract

Pruning decision trees is a useful technique for improving the generalization performance in decision tree induction, and for trading accuracy for simplicity in other applications. In this paper, a new algorithm called OPT-2 for optimal pruning of decision trees is introduced. The algorithm is based on dynamic programming. In its most basic form, the time and space complexities of OPT-2 are both $\Theta(nC)$, where n is the number of test nodes in the initial decision tree, and C is the number of leaves in the target (pruned) decision tree. This is an improvement over the recently published OPT algorithm of Bohanec and Bratko (which is the only known algorithm for optimal decision tree pruning) especially in the case of heavy pruning and when the tests of the given decision tree have many outcomes. If so desired, the space required by OPT-2 can further be reduced by a factor of r at the cost of increasing the execution time by a factor that is bounded above by $(r+1)/2$ (this is a considerable overestimate, however). From a practical point of view, OPT-2 enjoys considerable flexibility in various aspects, and is easy to implement.

1. Introduction

Pruning a decision tree, *DT*, is the process of replacing some of the subtrees of *DT* by leaves. In machine learning research, pruning is a widely used technique for avoiding overfitting of the training data. It is well known that, in noisy domains, pruning usually leads to decision trees with better generalization performance.

Another important motivation of pruning is "trading accuracy for simplicity" as discussed by Bohanec and Bratko [1]. They note that, in many situations, a simple,

* E-mail: hussein@ccse.kfupm.edu.sa.

SSDI 0004-3702(95)00060-7

comprehensible, but only approximate description of a concept may be more useful than a completely accurate description that involves a lot of details. They illustrate this idea by considering a rule that decides the legality of a white-to-move chess position (the KRK concept of [5]). They show that in representing this rule, (i) a decision tree with 11 leaves gives a completely accurate description, (ii) a pruned tree with only 4 leaves is 98.45% accurate, and (iii) a pruned tree with only 5 leaves is 99.57% accurate. Thus, more than half of the size of the completely accurate tree accounts for less than 0.5% of the accuracy. Pruning is helpful in such situations when the goal is to produce a compact concept description that is sufficiently accurate.

In the context of pruning, it is convenient to view the initial decision tree as a completely accurate one. The *accuracy* of a pruned decision tree then only indicates how close the pruned tree is to the initial tree.[1] A common practice when pruning a given decision tree, DT, with s leaves is to progressively replace various subtrees of DT by leaves leading to a sequence

$$DT_{s-1}, DT_{s-2}, \ldots, DT_1$$

of pruned decision trees such that each DT_i has at most i leaves. Viewing the *error* as the difference from the original tree DT, such trees are of increasing error. In an actual application, the *best* tree is to be selected from the above sequence of pruned trees based on some appropriate criteria, where the goal is to strike a good balance between the size of the tree and its accuracy.

In *optimal* pruning, it is required that the error of each DT_i in the sequence be minimum over all pruned trees of i (or less) leaves. Although decision tree pruning has been studied by many researchers (e.g., [2,4,6]), the only work that addressed optimal pruning so far is the recent paper of Bohanec and Bratko [1]. In their work, they show that previous methods lead to suboptimal solutions, and introduce an algorithm (which they call OPT) that guarantees optimality. Based on dynamic programming, OPT produces the above sequence of pruned trees in time $O(s^2)$, where s is the number of leaves of the initial decision tree.

This paper introduces a new algorithm, OPT-2, that also performs optimal pruning. An important feature of this new algorithm is its considerable flexibility. Unlike OPT, which generates the *whole* sequence of pruned trees *simultaneously*, the new algorithm works *sequentially* generating the trees of the sequence one after the other in increasing order of the number of leaves—that is, in the order $DT_1, DT_2, DT_3, \ldots$. One can argue that generating the whole sequence is often unnecessary, since, eventually, only one tree will be chosen from the sequence as the final result of pruning. Using OPT-2 gives the user the freedom to terminate the sequence generation quite early (and thus, saving unneeded computations) as soon as a tree that satisfies the criteria at hand is found.

Given that trees $DT_1, DT_2, \ldots, DT_{i-1}$ have already been generated, OPT-2 finds DT_i in time $\Theta(n)$, where n is the number of internal nodes (tests) of the initial decision tree. Thus, if the number of leaves of the *target* tree is C, then the total running time of OPT-2 will be $\Theta(nC)$. Usually, C is much smaller than s, since the goal is to prune

[1] Of course, under other definitions of accuracy (e.g. generalization performance in inductive learning), a pruned tree may even be more accurate than the initial tree.

the tree. Moreover, n is also smaller than s especially in the case of tests with many outcomes (namely, $s = (k-1)n+1$ in a k-ary tree). Therefore, OPT-2 is usually more efficient than OPT in terms of execution time. In the extreme case, however, when the whole sequence of pruned decision trees has to be generated, and when the tests of the decision tree are binary, OPT-2 will exhibit the same time complexity as OPT.

OPT-2 is also efficient in terms of space complexity. In its simplest form, the space required by the algorithm is $\Theta(nC)$. However, if one is willing to trade time for space, the required space can easily be reduced by a factor of r at the cost of increasing the execution time by a factor of much less than $(r+1)/2$. The space complexity of OPT-2 can be reduced in this manner down to $\Theta(nd_{\max})$, where $d_{\max}$ is the maximum degree [2] of the tree. The exact space complexity of OPT, on the other hand, is not clear. It can best be upper bounded by $O(s^2)$ although it may actually require a little less space than that [Bohanec, Private Communication, August, 1994].

In addition to the above advantages, the OPT-2 algorithm is also very easy to implement since (as will be seen later) it only involves simple computations over a two-dimensional array of positive numbers.

Although both OPT and OPT-2 are based on dynamic programming, the two algorithms differ substantially in the way the problem is divided into subproblems. The approach adopted in OPT is usually viewed as a "bottom-up" approach [3]. The idea there is to compute solutions for the subtrees rooted at the lowest level of the tree. Then from these, solutions at the nodes of the next upper level are computed. This process is repeated until the root of the initial tree is reached. A fundamental property of this bottom-up approach is that the trees DT_i of the pruning sequence are computed simultaneously while we are approaching the root. None of these pruned trees is final unless we reach the root, but once we are at the root the whole sequence becomes available at once. This consequence of the bottom-up approach is not desirable since it is rarely the case that the whole pruning sequence is needed.

In contrast, the OPT-2 algorithm enjoys more flexibility in that it produces the pruned trees one after the other, enabling the user to stop the process at any point as desired. This algorithm is a derivative of a dynamic programming method given by Johnson and Niemi for solving "tree knapsack" problems [3]. As will be seen from the description of the algorithm, processing in OPT-2 is done in a "left-to-right" fashion. Given that we have already computed trees $DT_1, DT_2, \ldots, DT_{i-1}$, and that necessary intermediate results from these computations are kept in memory, the OPT-2 algorithm finds DT_i from these through a linear left-to-right pass over the tree. This left-to-right approach, therefore, makes it possible to terminate the process once a satisfactory tree, say DT_j, is found—avoiding unneeded computation of trees DT_{j+1} through DT_{s-1}.

The rest of this paper is organized as follows. We start in the next section with a formal description of the problem of optimal pruning of decision trees. The OPT-2 algorithm is then described in Sections 3 and 4. Space and time complexity analyses are given in Sections 5 and 6. We conclude with a summary and suggestions for future research in Section 7.

[2] The degree of a test node is the number of its children.

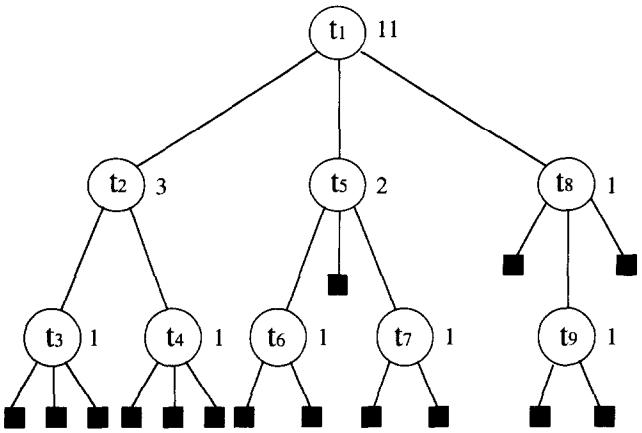

Fig. 1. A decision tree (from [1]) with error measurement given at each test node. Error is assumed to be 0 at all the leaves of this decision tree.

2. The problem

Let DT be a decision tree. Each internal node in DT is labeled with a test t_i, and each outgoing edge from that node corresponds to a possible outcome of that test. Each leaf is labeled with some *class*. The class at a given leaf indicates the "action" to be taken for cases that satisfy the tests along the branches from the root of the tree to that leaf.

For convenience, we will always assume that the test nodes are labeled $t_1, t_2, t_3, \ldots, t_n$ (where n denotes the number of test nodes) according to the depth-first traversal of the tree. (See Fig. 1.) The *degree* of a node t_i, denoted $d(t_i)$, is the number of children of t_i.

For a test node t_i, let E_i be the set of cases that satisfy the tests along the branches from the root to t_i. The *majority class* at t_i is the most frequent class in E_i. *Pruning* a test node t_i means replacing the subtree of DT rooted at t_i by a leaf labeled with the majority class at t_i. A decision tree DT' generated by pruning one or more nodes from DT in this manner is a *predecessor-closed* subtree of DT in the sense that for each test node $t_j \in DT'$, the parent of t_j is also in DT'.

By $error(t_i)$, we denote the number of cases in E_i that are incorrectly classified if the test node t_i is pruned. In other words, if a is the number of cases in E_i labeled with the majority class, then $error(t_i) = E_i - a$.[3] By $error(DT)$, we denote the number of cases that are incorrectly handled by DT. This is equivalent to $\sum_{t \in \ell} error(t)$, where ℓ is the set of the leaves of DT. The *size* of DT, denoted $s(DT)$, is the number of leaves in DT. The *base-error* is the error when the whole tree is replaced with a single leaf. Thus, the *base-error* is just $error(t_1)$.

[3] As far as the presented algorithm is concerned, the *error* can be any non-negative value (not necessarily the error frequency) such that the error at a given node is greater than or equal to the sum of the errors at the children of that node.

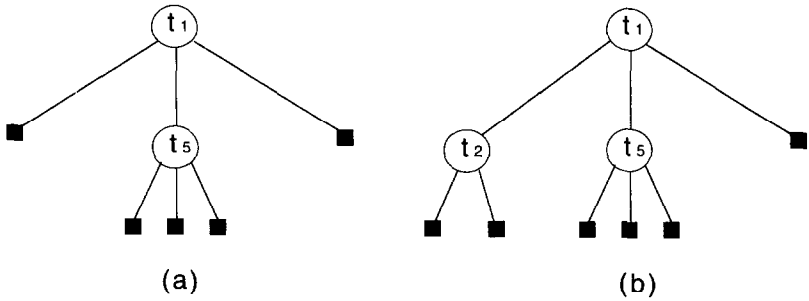

Fig. 2. Expanding a leaf in a pruned decision tree.

The problem of optimal pruning can be stated as an optimization problem as follows:

> Given a decision tree DT and a positive integer C, find a pruned decision tree DT' from DT such that $s(DT) \leqslant C$ and $error(DT)$ is minimized.

As in [1], we assume that the error at each test node is given (see Fig. 1). Obviously, solving the problem as stated above is sufficient to generate the pruning sequence as mentioned in the previous section.

3. The method

Pruning can be viewed as deciding for each node t_i in the initial decision tree whether or not that node is to be included in the final tree. Consider the decision tree of Fig. 2(a) which is generated by pruning the decision tree of Fig. 1 at nodes t_2, t_6, t_7 and t_8. Assume that, in addition to the test nodes of this tree, we decided to include node t_2 as well. This leads to the tree shown in Fig. 2(b) in which the left-most child of t_1 has been replaced by node t_2 with two leaves as its children. This move, which can be viewed as "leaf-expansion", reduces the error of the tree from 6 to 5. The number of the leaves, however, increases from 5 to 6. Similarly, if we further choose to expand, for example, the leaf appearing as the right child of t_2 in Fig. 2(b), then the error will decrease to 4, but the total number of leaves will now be 8.

In this paper, the reduction in error which results from expanding a leaf is viewed as "profit", while the associated increment in the size of the tree is viewed as "cost". Now, given a decision tree DT that we would like to prune, consider generating another tree, T, by doing the following:

- Add a new node, t_0, as the parent of t_1 (the root of DT). This new node, thus, becomes the root of T.
- For each node t_i in T, $1 \leqslant i \leqslant n$, let

$$cost(t_i) = d(t_i) - 1$$

and let $cost(t_0) = 1$.

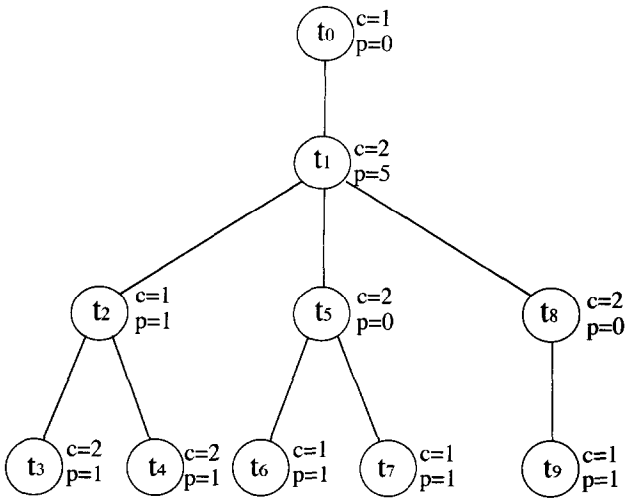

Fig. 3. A cost–profit tree generated from the decision tree of Fig. 1. At each node, c and p indicate the cost and profit, respectively.

- For each node t_i in T, $1 \leqslant i \leqslant n$, let

$$profit(t_i) = error(t_i) - \sum_{a:\ \text{child of } t_i} error(a)$$

  and let $profit(t_0) = 0$.
- Delete all the leaves of DT.

We will call the tree T constructed as above the *cost–profit (CP) tree* of DT. Fig. 3 shows the CP tree that corresponds to the decision tree of Fig. 1.

Note that there is a one-to-one correspondence between the set of all pruned subtrees of a decision tree DT, and the set of all predecessor-closed subtrees of the CP tree T obtained from DT. Namely, a predecessor-closed subtree, T', of the CP tree T corresponds to the pruned decision tree DT' in which each subtree of DT rooted at a node $t_i \notin T'$ is replaced by a leaf.

For a predecessor-closed subtree T', let us denote by $cost(T')$, the total cost, and by $profit(T')$, the total profit of the nodes of T'. The reader may confirm that if DT' is the pruned decision tree which corresponds to T', then

- $cost(T') = s(DT')$, and
- $profit(T') = base\text{-}error - error(DT')$.

As an example, let T' be the subtree of the cost–profit tree shown in Fig. 3 induced by the nodes $\{t_0, t_1, t_2, t_4, t_5\}$. The corresponding pruned decision tree DT' in this case is obtained by replacing the test nodes $\{t_3, t_6, t_7, t_8\}$ in the decision tree of Fig. 1 by leaves. The cost of T' is 8, which is equivalent to the number of leaves in DT'. The profit of T' is 7. This is equivalent to the difference between the *base-error*, which is 11, and the error of DT', which is 4.

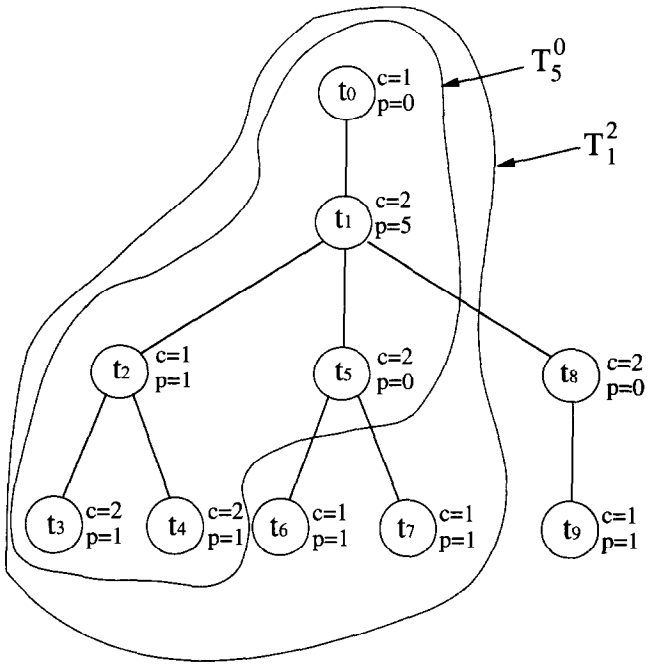

Fig. 4. Examples of T_i^j subtrees.

With the above transformation from a decision tree DT to a CP tree T, the optimal pruning problem addressed here can be restated as follows:

> Given a CP tree T, and a positive integer C, find a predecessor-closed subtree T' of T such that $profit(T')$ is maximized and $cost(T') \leqslant C$.

Notation. For a node t_i in a CP tree T and an integer j, $0 \leqslant j \leqslant d(t_i)$, let us denote by T_i^j the subtree of T induced by

(1) the nodes $t_0, t_1, t_2, t_3, \ldots, t_{i-1}, t_i$ (which are the nodes visited in a depth-first traversal of the tree up to t_i), and
(2) the first j children of t_i and all their successors.

Fig. 4 shows two examples. T_5^0 is the subtree induced by node t_5 and all nodes of lower indices. None of the children of t_5 is included because the superscript in T_5^0 is 0. Likewise, T_1^2 is the subtree induced by node t_1, its first two children and their successors (which are $\{t_2, t_3, t_4, t_5, t_6, t_7\}$), and node t_0 since it has a lower index.

Notation. Note that any T_i^j is itself a cost–profit tree. For some positive integer c and some T_i^j, let $Solution[T_i^j, c]$ denote a maximum profit predecessor-closed subtree of T_i^j *that contains* t_i and whose cost is less than or equal to c. By $P[T_i^j, c]$ we will denote the profit of that solution, i.e., $profit(Solution[T_i^j, c])$.

For example, consider the subtree T_1^2 of Fig. 4. If the maximum allowed cost is 7, then the subtree induced by nodes $\{t_0, t_1, t_5, t_6, t_7\}$ gives the highest possible profit which is 7. Therefore, $P[T_1^2, 7] = 7$ in this case.[4]

With the above notation, our goal becomes to find $Solution[T_0^1, C]$, where C is the target tree size. It should be noted that, even though some subtrees such as T_1^2 and T_5^2 are equivalent, a solution for T_5^2 has to include t_5 whereas a solution for T_1^2 does not. The reader may check, for example, that $P[T_1^2, 6]$ is 7 while $P[T_5^2, 6]$ is only 6.

Normally, $P[T_i^j, c]$ takes non-negative values. However, since $Solution[T_i^j, c]$ has to contain t_i and all its predecessors (this is because the solution subtree has to be predecessor-closed), no solution exists if c is less than the total cost of these nodes. In this case, we let $P[T_i^j, c]$ be $-\infty$. For example, for T_5^0 of Fig. 4, there exists no solution of cost 4, and thus, $P[T_5^0, 4] = -\infty$.

The pruning algorithm introduced in this paper is based on the following three rules:

- *Rule* 1:

$$P[T_0^0, c] = 0.$$

- *Rule* 2: Let t_i be the kth child of t_r. Then

$$P[T_i^0, c] = \begin{cases} P[T_r^{k-1}, c - cost(t_i)] + profit(t_i), & \text{if } c - cost(t_i) \geqslant 1, \\ -\infty, & \text{otherwise.} \end{cases}$$

- *Rule* 3: Let t_i's jth child be t_k. Then

$$P[T_i^j, c] = \max\left\{P[T_k^{d(t_k)}, c], P[T_i^{j-1}, c]\right\}.$$

The correctness of these rules is not difficult to confirm:

- Rule 1 is trivial.
- For Rule 2, the situation is depicted in Fig. 5. Note that T_i^0 contains no children of t_i, and that, by definition, a solution for T_i^0 must contain t_i. Therefore, given that t_i is the child number k of t_r, a solution within cost c has to be the node t_i along with a solution for T_r^{k-1} within cost $c - cost(t_i)$. But, of course, if $c - cost(t_i) \leqslant 0$, then no solution exists for T_i^0 within cost c, and thus, $P = -\infty$ in this case.
- For Rule 3, note that both T_i^j and $T_k^{d(t_k)}$ denote the same subtree. However, by definition, any solution for $T_k^{d(t_k)}$ has to include t_k. To find a solution for T_i^j, we have two choices, either to include t_k (possibly with some of its descendants) or not to include it. Not including t_k means just finding a solution for T_i^{j-1} (the tree T_i^j with t_k and all its descendants removed), while including t_k means finding a solution for $T_k^{d(t_k)}$. The rule states that we are to choose one of these two solutions, whichever gives higher profit.

Our goal, that of computing $Solution[T_0^1, C]$ for some target tree size C, can be achieved by repeated application of the above rules. Even though the above rules only compute the profit of the solution for a given cost (and not the solution itself), it turns

[4] Note that other solutions with the same profit also exist. When there are multiple solutions, finding one will be considered sufficient in this work.

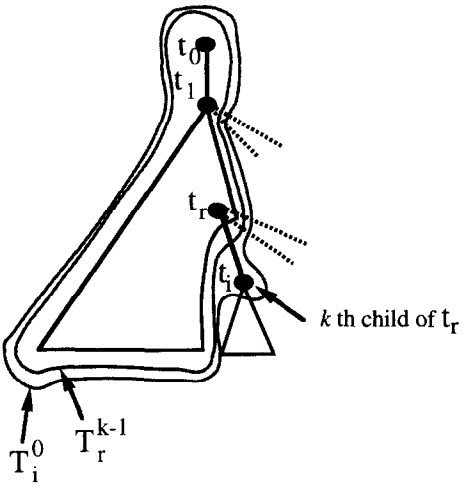

Fig. 5. A situation where Rule 2 applies.

out that the actual solution can be easily computed from these numbers. This will be explained in the next section.

To be able to apply the rules, the computations have to be done in a proper order so that the P values on the right-hand side of a rule are computed before we attempt to apply that rule. For this purpose, we order the subtrees T_i^j's of T such that

(1) T_i^j precedes T_i^{j+1}.

(2) If t_k is the jth child of t_i, then T_i^{j-1} precedes T_k^0 and $T_k^{d(t_k)}$ precedes T_i^j.

For example, the T_i^j's of the CP tree shown in Fig. 3 are ordered as $T_0^0, T_1^0, T_2^0, T_3^0, T_2^1, T_4^0, T_2^2, T_1^1, T_5^0, T_6^0, T_5^1, T_7^0, T_5^2, T_1^2, T_8^0, T_9^0, T_8^1, T_1^3, T_0^1$.

Notation. $T(k)$ will be used to denote the subtree number k in the above-mentioned ordering. Moreover, for any $j \geqslant 1$, T_i^{j-1} will be called the *elder sibling* of T_i^j. If T_i^{j-1} is the kth subtree, and T_i^j is the yth subtree in the ordering, then we write $ES(y) = k$. It is obvious that $ES(y)$ is always less than y.

For the CP tree of Fig. 3, $T(3) = T_2^0$, $T(5) = T_2^1$ and $T(7) = T_2^2$, and thus, $ES(5) = 3$ and $ES(7) = 5$.

Note that, for any $i \geqslant 1$, if T_i^0 is the yth subtree in our ordering, and t_i is the kth child of t_r, then T_r^{k-1} is the $(y-1)$th subtree in the ordering. Therefore, using the above notation, Rule 2 can be restated as follows:

- *Rule* 2: If $T(y) = T_i^0$ for $1 \leqslant i \leqslant n$, then

$$P[T(y), c] = \begin{cases} P[T(y-1), c - cost(t_i)] + profit(t_i), \\ \qquad \text{if } c - cost(t_i) \geqslant 1, \\ -\infty, \quad \text{otherwise.} \end{cases}$$

Table 1
The P values for the CP tree given in Fig. 3; "$*$" indicates $-\infty$; ES indicates the *elder sibling* information; the underlined values are the values referenced when constructing a solution of total cost 10

y	$T(y)$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ES
1	T_0^0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	·
2	T_1^0	*	*	5	5	5	5	5	5	5	5	5	5	5	5	5	·
3	T_2^0	*	*	*	6	6	6	6	6	6	6	6	6	6	6	6	·
4	T_3^0	*	*	*	*	*	7	7	7	7	7	7	7	7	7	7	·
5	T_2^1	*	*	*	6	6	7	7	7	7	7	7	7	7	7	7	3
6	T_4^0	*	*	*	*	*	7	7	8	8	8	8	8	8	8	8	·
7	T_2^2	*	*	*	6	6	7	7	8	8	8	8	8	8	8	8	5
8	T_1^1	*	*	5	6	6	7	7	8	8	8	8	8	8	8	8	2
9	T_5^0	*	*	*	*	5	6	6	7	7	8	8	8	8	8	8	·
10	T_6^0	*	*	*	*	*	6	7	7	8	8	9	9	9	9	9	·
11	T_5^1	*	*	*	*	5	6	7	7	8	8	9	9	9	9	9	9
12	T_7^0	*	*	*	*	*	6	7	8	8	9	9	10	10	10	10	·
13	T_5^2	*	*	*	*	5	6	7	8	8	9	9	10	10	10	10	11
14	T_1^2	*	*	5	6	6	7	7	8	8	9	9	10	10	10	10	8
15	T_8^0	*	*	*	*	5	6	6	7	7	8	8	9	9	10	10	·
16	T_9^0	*	*	*	*	*	6	7	7	8	8	9	9	10	10	11	·
17	T_8^1	*	*	*	*	5	6	7	7	8	8	9	9	10	10	11	15
18	T_1^3	*	*	5	6	6	7	7	8	8	9	9	10	10	10	11	14
19	T_0^1	0	0	5	6	6	7	7	8	8	9	9	10	10	10	11	1
Error		11	11	6	5	5	4	4	3	3	2	2	1	1	1	0	

Likewise, for $j \geqslant 1$, if T_i^j is the yth subtree, and t_k is the jth child of t_i, then $T_k^{d(t_k)}$ is just $T(y-1)$, and T_i^{j-1} is just $T(ES(y))$. Therefore, Rule 3 can be restated as follows:

- *Rule* 3: If $T(y) = T_i^j$ for some $j \geqslant 1$, then

$$P[T(y), c] = \max\{P[T(y-1), c], P[T(ES(y)), c]\}.$$

The reader can confirm that computing $P[T(y), c]$ can be done by two nested loops for $c = 1, 2, 3, \ldots$ as the outer loop and for $y = 1, 2, 3, \ldots$ as the inner one. Thus, performing the computation in the above sequence not only enables the use of the rules, but also makes the algorithm quite easy to implement.

3.1. Example

Table 1 shows all the P values for the CP tree of Fig. 3. For a given target tree size C, columns are computed from left to right for $c = 1, 2, \ldots, C$. Within each column, computation is done from top to bottom. When computing $P[T_i^j, c]$, Rule 1 is applied if $i = j = 0$, and Rule 2 is applied if $i \neq 0$ but $j = 0$. Rule 3 is applied otherwise.

For Rule 2, suppose T_i^0 is the yth subtree in our ordering. To compute $P[T_i^0, c]$ (which is the entry on row y, column c), all we need to do is to add $profit(t_i)$ to the

entry at row $y-1$ and column $c - cost(t_i)$. For example, let us see how to compute the entry at row 6, column 9, which is $P[T_4^0, 9]$. The profit of t_4 is 1 and the cost is 2. Therefore, this entry is given by $P[T(5), 9-2]+1=8$.

For Rule 3, suppose T_i^j is the yth subtree in our ordering. To compute $P[T_i^j, c]$ (which is the entry on row y, column c), we need to compare the entries $P[T(y-1), c]$ and $P[T(ES(y)), c]$, and take the larger of these. As an example, let us compute the entry at row 18, column 10, which is $P[T_1^3, 10]$. The first term is $P[T(17), 10] = 8$. Since $ES(18) = 14$, the second term is $P[T(14), 10] = 9$. Taking the maximum of these, $P[T(18), 10]$ thus becomes 9.

Computation of the P values is terminated after computing column C and $P[T_0^1, C]$ (the value at the bottom of column C) is taken as the final solution. For example, when the target tree size is 10, the maximum profit is 9. The row at the bottom of Table 1 indicates the error of the corresponding pruned tree. It says, in this case, that the best tree of size 10 has error

$$base\text{-}error - P[T_0^1, 10] = 11 - 9 = 2.$$

4. Constructing actual solutions

Once the P values have been computed up to column C, where C is the target tree size, the following three rules can be applied to find $Solution[T_i^j, c]$ for any T_i^j and $c \leqslant C$:

- *Rule* 1′:

$$Solution[T_0^0, c] = \{t_0\}.$$

- *Rule* 2′: Let T_i^0 be the yth subtree in the ordering. Then

$$Solution[T(y), c] = \begin{cases} \{t_i\} \cup Solution[T(y-1), c - cost(t_i)], \\ \qquad\qquad \text{if } P[T(y), c] \geqslant 0, \\ \text{no solution exists,} \quad \text{otherwise.} \end{cases}$$

- *Rule* 3′: Let T_i^j be the yth subtree in the ordering. Then

$$Solution[T(y), c] = \begin{cases} Solution[T(y-1), c], & \text{if } P[T(y), c] = P[T(y-1), c], \\ Solution[T(ES(y)), c], & \text{otherwise.} \end{cases}$$

The correctness of these rules can be verified following the same logic of the three rules of the previous section.

4.1. Example—revisited

For target tree size 10, let us see how we can actually construct a minimal error tree within this size after the first 10 columns of Table 1 have been computed. Our

$$Solution[T(19),10] = Solution[T(18),10] = Solution[T(14),10] = Solution[T(13),10] = Solution[T(12),10] = \{t_7\} \cup Solution[T(11),9] = \{t_7\} \cup Solution[T(10),9] = \{t_7, t_6\} \cup Solution[T(9),8] = \{t_7, t_6, t_5\} \cup Solution[T(8),6] = \{t_7, t_6, t_5\} \cup Solution[T(7),6] = \{t_7, t_6, t_5\} \cup Solution[T(6),6] = \{t_7, t_6, t_5, t_4\} \cup Solution[T(5),4] = \{t_7, t_6, t_5, t_4\} \cup Solution[T(3),4] = \{t_7, t_6, t_5, t_4, t_2\} \cup Solution[T(2),3] = \{t_7, t_6, t_5, t_4, t_2, t_1\} \cup Solution[T(1),1] = \{t_7, t_6, t_5, t_4, t_2, t_1, t_0\}.$$

Fig. 6. Computing $Solution[T_0^1, 10]$.

```
Algorithm OPT-2(CPTree,C)
local variable c: integer
begin
  Y = 0
  DepthFirstPreprocessing(CPTree)
  c = 0
  repeat until c = C
    c = c + 1
    ComputeP(c)
  FindActualSolution(C)
end
```

Fig. 7. Main routine of the OPT-2 algorithm.

goal is, thus, to compute $Solution[T_0^1, 10] = Solution[T(19), 10]$. This can be done using the three rules given above as follows: (i) By Rule 3′, since $P[T(19), 10] = P[T(18), 10]$, the desired solution is equal to $Solution[T(18), 10]$. (ii) By Rule 3′, since $P[T(18), 10] > P[T(17), 10]$ and since $ES(18) = 14$, the desired solution is equal to $Solution[T(14), 10]$. (iii) By Rule 3′, since $P[T(14), 10] = P[T(13), 10]$, the solution is again equal to $Solution[T(13), 10]$. (iv) Again, $P[T(13), 10] = P[T(12), 10]$ and thus the desired solution is just $Solution[T(12), 10]$. (v) $T(12) = T_7^0$, and therefore, Rule 2′ is the one to be applied this time (since the superscript is 0). Given that $cost(t_7) = 1$, the desired solution is the union of $\{t_7\}$ and $Solution[T(11), 9]$. Continuing in this manner (see Fig. 6), we find that the final solution is $\{t_7, t_6, t_5, t_4, t_2, t_1, t_0\}$.

5. Time complexity

A formal description of the OPT-2 algorithm is shown in Figs. 7, 8, 9 and 10. All the variables are assumed global unless declared otherwise. A CP tree is assumed given in which nodes are labeled $t_0, t_1, t_2, \ldots, t_n$ in depth-first ordering. Each node t_i has cost and profit values accessible by calling `cost(i)` and `profit(i)`.

```
Procedure DepthFirstPreprocessing(Tree)
local variable es, i: integer
begin
  Y = Y + 1
  NodeOf[Y] = root of Tree
  ChildNo[Y] = 0
  es = Y  /* es is a local variable !! */
  for i = 1 to d(root of Tree)  /* for all children of root */
      DepthFirstPreprocessing(i-th subtree of Tree)
      Y = Y + 1
      NodeOf[Y] = root of Tree
      ChildNo[Y] = i
      ElderSibling[Y] = es
      es = Y
end
```

Fig. 8. Initialization routine.

```
Procedure ComputeP(c)
local variable y: integer
begin
  for y = 1 to Y
      if NodeOf[y] = 0 and ChildNo[y] = 0 then P[y,c]=0  /*Rule1*/
      else if ChildNo[y] = 0 then                        /*Rule2*/
        if c - cost(NodeOf[y]) > 0 then
           P[y,c] = P[y-1,c-cost(NodeOf[y])]+profit(NodeOf[y])
        else P[y,c] = - infinity
      else P[y,c] = max{P[y-1,c],P[ElderSibling[y],c]}   /*Rule3*/
end
```

Fig. 9. Computation of the P values.

The two arrays NodeOf[] and ChildNo[] store the subtree information. If subtree T_i^j is the yth subtree in the ordering, then NodeOf[y] = i and ChildNo[y] = j. The two-dimensional array P[,] is used to store the P values, as illustrated in Table 1. Thus, entry P[y,c] is just $P[T(y), c]$.

The array ElderSibling[] is used to store the elder sibling information. Thus, if T_i^{j-1} is the xth subtree and T_i^j is the yth array in the ordering, we let ElderSibling[y] = x.

As shown in Fig. 7, the algorithm starts by calling the initialization procedure DepthFirstPreprocessing (Fig. 8). This procedure initializes the arrays NodeOf[], ChildNo[] and ElderSibling[] by performing a depth-first traversal of the given CP tree. The algorithm then starts computing the P[y,c] entries for increasing values of c until some stopping criterion is reached. The computation of P[y,c] is done by the procedure ComputeP(c) (Fig. 9), which implements the three rules given in Sec-

```
Procedure FindActualSolution(c)
local variable y: integer
begin
  y = Y
  if P[y,c] < 0 then
     output('No Solution')
  else
  while y > 1 do
     if ChildNo[y] = 0 then
        output(NodeOf[y])
        c = c - cost(NodeOf[y])
        y = y - 1
     else if P[y,c] = P[y-1,c] then y = y - 1
          else y = ElderSibling[y]
end
```

Fig. 10. Computing actual solutions.

tion 3. Actual solutions can be generated from the array `P[,]` by calling the procedure `FindActualSolution`.

Let us now analyze the time complexity of OPT-2. Let n and s be the number of internal nodes (tests) and the number of leaves (respectively) in the original decision tree. The number of nodes in the corresponding cost–profit tree is then $n + 1$. The number of leaves, C, in the target decision tree, can be any integer between 1 and s. In most cases, however, C is significantly smaller than s since the goal is to prune the tree.

The number of T_i^j's (which is the variable `Y` in the algorithm) is exactly $2n + 1$. The procedure `DepthFirstPreprocessing` simply fills in the entries `NodeOf[y]`, `ChildNo[y]` and `ElderSibling[y]` for y from 1 up to `Y`, and thus, this initialization step runs in time $\Theta(n)$.

The time needed to execute each iteration in `ComputeP()` is bounded by a constant. Therefore, a call to `ComputeP()` runs in time $\Theta(n)$.

Finally, the last step of computing the actual solution obviously takes time $\Theta(n)$ in the worst case. Summing up, the overall running time of the algorithm is $\Theta(nC)$.

6. Space complexity

It is interesting to note that, unlike OPT, actual solutions in OPT-2 are never stored explicitly in memory but only computed when needed. The space required by the algorithm as described so far is dominated by the array `P[,]` whose size is $\Theta(nC)$. This is usually reasonable for practical purposes. However, in some situations, reducing the amount of space is given high priority even if it may cause some increase in the execution time. Trading time for space can easily be achieved in OPT-2. The idea is based on the following observations:

- Given that the maximum degree over all the nodes of the initial decision tree is $d_{\max}$, the maximum cost over all the nodes of the corresponding CP tree is $d_{\max} - 1$. Therefore, applying Rule 2 when computing say column number c of the `P[,]` matrix never requires accessing any of the columns 1 through $c - d_{\max}$.
- Applying Rule 3 when computing column number c is done by accessing the P values in column c only.

We can, therefore, implement the `P[,]` matrix as a "circular array" keeping only the *most recent* $d_{\max}$ columns, and thus, making the required space as low as $O(nd_{\max})$. Constructing actual solutions will, however, need some of the "forgotten" P values, and thus, these have to be recomputed. Consider now the following observation:

- When computing *Solution*$[T(y), c]$, all the P values needed are in the portion of the `P[,]` matrix confined between `P[1,1]` and `P[y-1,c]`.

These observations suggest the following strategy: Let k be any fixed integer greater than or equal to $d_{\max}$. We start by computing the P matrix column by column such that when the cth column is being computed, only the k most recent columns (i.e., the columns $c, c-1, \ldots, c-k$) are kept in memory. When we reach the target tree size C, we trace the P values back in order to construct an actual solution (as done in the procedure `FindActualSolution`). Suppose that, during this trace, an unavailable entry, say `P[y1,c1]`, such that `c1` $< c - k$, is needed. At this point, we stop the tracing temporarily and start recomputing the `P[,]` matrix, but this time only up to row `y1` -1, and column `c1`. Again, only the most recent k columns are kept during the computation. We then continue tracing back to construct the solution. Tracing and recomputing is done alternately in this manner until the whole solution is generated.

For example, consider again the P values given in Table 1. Let k be 5, and suppose the target tree size is 10. For this case, we first compute the columns 1 through 10 of the table. However, at the time we finish computing column 10, only columns 6, 7, 8, 9, and 10 are available in memory. In order to find *Solution*$[T(19), 10]$, we trace the P values back as shown in Fig. 6. After generating some part of the solution (namely, the set $\{t_7, t_6, t_5, t_4\}$), we will need to access the entry `P[5,4]`. This entry is not available, so we recompute the area of the `P[,]` matrix confined between the entry `P[1,1]` and `P[5,4]`. We then continue tracing back as done in Fig. 6 until the whole solution is generated.

In the above example, the required space is reduced by a factor of 2, while the number of P values that have to be computed increases from 190 to 210. If we let k be 3 for this same example, then recomputation will occur once at the entry `P[8,6]` and again at the entry `P[3,2]`. In this case, the space is cut roughly by a factor of 3, while the number of P value computations increases from 190 to 244.

It can be verified that if we choose to reduce the required space by a factor of r (that is to let $k \approx C/r$, provided that $k \geqslant d_{\max}$), then the increment in the execution time is bounded by a factor of $(r+1)/2$. This is, however, a considerable overestimate.

Thus, based on the available resources, the user of OPT-2 can freely adjust the space complexity of the algorithm at any level between $\Theta(nC)$ and $\Theta(nd_{\max})$. The ability to trade time for space and vice versa in this manner gives the user of OPT-2 considerable flexibility to meet his/her requirements.

7. Summary and future research

In this paper, a new algorithm, OPT-2, for optimal pruning of decision trees was introduced. The algorithm is based on dynamic programming. In its most basic form, OPT-2 runs in time $\Theta(nC)$, and requires space $\Theta(nC)$, where n is the number of test nodes in the initial decision tree, and C is the number of leaves in the target (pruned) decision tree. If so desired, the space required by the algorithm can be reduced by a factor of r at the cost of increasing the execution time by a factor that is bounded above by $(r+1)/2$ (this is a considerable overestimate, however). OPT-2 is, therefore, an improvement over the recently published OPT algorithm of Bohanec and Bratko (which is the only known algorithm for optimal decision tree pruning) especially in the case of heavy pruning and when the tests of the given decision tree have many outcomes. Moreover, from a practical point of view, OPT-2 is quite flexible and easy to implement.

An interesting problem for future research is to consider pruning without imposing the requirement of "predecessor-closedness" over the pruned decision trees. In this setting, pruning means that a subtree of the initial decision tree may possibly be replaced by one of its subtrees, and not necessarily by a leaf. The only work that considers this kind of pruning is Quinlan's C4.5 package [7]. However, pruning is done only heuristically in that work. It is interesting to design a polynomial time algorithm for optimal pruning when pruning is defined in this way and/or to prove results on computational hardness of such a task.

Acknowledgements

The author thanks King Fahd University of Petroleum and Minerals for supporting this research. Most of this work was done when the author was visiting NTT Communication Science Laboratories of Nippon Telegraph and Telephone Corporation, Japan. He thanks Dr. Shigeo Kaneda and the machine learning group at NTT for useful discussions and support. The author is also grateful to Marko Bohanec for valuable comments and for kindly clarifying various aspects of the OPT algorithm.

References

[1] M. Bohanec and I. Bratko, Trading accuracy for simplicity in decision trees, *Mach. Learn.* **15** (1994) 223–250.

[2] L. Breiman, J.H. Friedman, R.A. Olshen and C.J. Stone, *Classification and Regression Trees* (Belmont, Wadsworth, 1984).

[3] D.S. Johnson and K.A. Niemi, On knapsacks, partitions and a new dynamic programming technique for trees, *Math. Oper. Res.* **8** (1983) 1–14.

[4] J. Mingers, An empirical comparison of pruning methods for decision tree induction, *Mach. Learn.* **4** (1989) 227–243.

[5] S. Muggleton, M. Bain, J. Hayes-Michie and D. Michie, An experimental comparison of human and machine learning formalisms, in: *Proceedings Sixth International Conference on Machine Learning*, Ithaca, NY (Morgan Kaufmann, San Mateo, CA, 1989).

[6] J.R. Quinlan, Induction of decision trees, *Mach. Learn.* **1** (1986) 81–106.

[7] J.R. Quinlan, *C4.5: Programs for Machine Learning* (Morgan Kaufmann, San Mateo, CA, 1989).