Subtree Replacement in Decision Tree Simplification

Salvatore Ruggieri
Dipartimento di Informatica, Università di Pisa, Italy
ruggieri@di.unipi.it

Abstract

The current availability of efficient algorithms for decision tree induction makes intricate post-processing techniques worth to be investigated both for efficiency and effectiveness. We study the simplification operator of subtree replacement, also known as *grafting*, originally implemented in the C4.5 system. We present a parametric bottom-up algorithm integrating grafting with the standard pruning operator, and analyze its complexity in terms of the number of nodes visited. Immediate instances of the parametric algorithm include extensions of error based, reduced error, minimum error, and pessimistic error pruning. Experimental results show that the computational cost of grafting is paid off by statistically significant smaller trees without accuracy loss.

1 Introduction

Decision tree induction has been extensively studied in the machine learning and data mining communities as a solution to the classification task. The common problems of over-fitting the training data and of "trading accuracy for simplicity" [2] have been addressed by a large class of post-processing algorithms called *simplification methods*. We refer the reader to [4, 6, 11, 21] for surveys and empirical comparisons.

The most well-known simplification method is decision tree *pruning*, which consists of turning a decision node into a leaf by discarding the whole subtree rooted at the node. Pruning algorithms typically proceed bottom-up, by relying on an error estimation function to compare the errors of the two alternatives: to prune a decision node or not. Pruning is not, however, the only simplification operator. The popular C4.5 system [20] adds to the error based pruning method the *grafting*[1] or *subtree replacement* operator. Grafting a decision node consists of replacing the subtree rooted at the father of the node by the subtree rooted at the node. Fig. 1 shows the effects of grafting node 3: its subtree replaces the father's subtree, and the subtree rooted at the sibling 4 is discarded. C4.5 considers grafting the *largest child* of a decision node, where the size of a child is the number of cases in the training set that follow the branch from the parent to the child node. The decision of whether to graft, to prune, or to leave a node unchanged is, again, made on the basis of the error estimates of the three alternatives. The underlying intuition in considering the largest child is that its subtree is likely to carry most of the discriminative power of the father's subtree. An immediate variant consists of considering any child node as a candidate for subtree replacement, by calculating the error estimates for each child and then choosing the one with the lowest estimate, or, in case of ties, the one with the smallest size. We call *grafting the largest child*, or simply grafting, the C4.5 approach and *grafting any child* the variant above. Grafting any child has been considered in the context of model trees [9] in conjunction with the reduced error estimation function, but separately from the pruning operator.

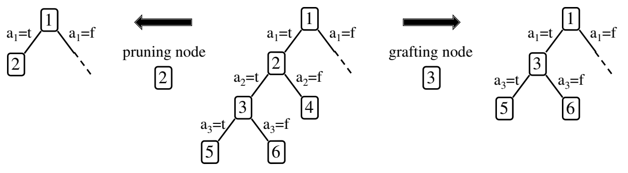

Figure 1: Pruning (left) and grafting (right) samples.

The study of simplification methods is a relevant problem in decision tree induction, where the objective is to produce models that trade off:

- simplicity and interpretability, typically measured by tree size, tree depth, or other structural metrics;
- with accuracy and generalizability of predictions, typically measured by error rate on unseen cases.

So far, the investigation of intricate simplification methods (i.e., other than node pruning) has been discouraged

[1]The term "grafting" was originally introduced by Esposito et al. [11]. Unfortunately, the term has been also used in the decision tree literature [26] for a different technique that *adds* leaves to a tree to improve its predictive accuracy. In this paper, we use the term in the former sense.

by time-consuming tree building algorithms: adding a second (possibly long) stage to tree induction has not been considered an option by researchers and practitioners. The current availability of efficient (sequential and parallel) implementations of decision tree induction algorithms [1, 14, 22, 24] makes post-processing techniques worth to be investigated both for efficiency and effectiveness. Nevertheless, grafting has not been adopted by decision tree induction systems other than C4.5. Why? In our opinion, the reason is threefold. First, *the integration of pruning and grafting algorithms is not well-understood.* The C4.5 implementation, for instance, is a cumbersome doubly-recursive procedure. Second, *grafting is perceived as computationally more expensive than pruning*, but the added computational cost has not been clearly stated. Third, *the effectiveness of grafting* in reducing tree size while not worsening accuracy *has not been assessed.* In this paper, we study grafting in conjunction with bottom-up pruning precisely to solve those three issues. As original contributions of this paper, *first*, we describe a general bottom-up algorithm integrating grafting and pruning which is parametric in an error estimation function. Instances of the algorithm conservatively extend error based pruning (EBP), reduced error pruning (REP), minimum error pruning (MEP), and pessimistic error pruning (PEP). *Second,* we characterize lower and upper bounds for the computational cost added by grafting to pruning, in terms of the number of nodes visited. This is a standard cost measure in tree and graph data structure analysis. *Third,* we experimentally compare over 14 medium and large public datasets the performances of pruning vs grafting the largest child vs grafting any child, with respect to elapsed time, error rate, tree size (as per number of nodes), and weighted depth (average depth of leaves in classifying test cases [19]) for the four above mentioned simplification methods.

The paper is organized as follows. Sect. 2 recalls background definitions, states the experimental methodology and lists the datasets used. In Sect. 3, we introduce a simplification algorithm which integrates pruning and grafting, and that is parametric in an error estimation function. In Sect. 4, we study its complexity and experimentally compare it to pruning. Related work is discussed in Sect. 5. Finally, Sect. 6 summarizes the contributions of the paper.

2 Background

2.1 Decision Tree Induction Let us state some basic definitions and notation. Consider a relation $\mathcal{T}$, called a *dataset*. Tuples in $\mathcal{T}$ are called *cases*. An attribute of the relation is the *class* attribute, while the remaining ones are called the *predictive* attributes. The domain of a predictive attribute can be discrete or continuous, with missing values allowed. The domain of the class is discrete, with no missing value in the dataset. Classifiers are induced from a subset of $\mathcal{T}$, the *training set*, and tested on the remainder of cases, the *test set*. The error rate on the test set is a measure of the quality of the classifier. In n-fold cross-validation, $\mathcal{T}$ is randomly partitioned into n folds of approximatively equal size. For each fold, a classifier is induced from the remaining folds and tested on it. In stratified n-fold cross-validation, each fold maintains the same proportion of class values as in $\mathcal{T}$.

A *decision tree* is a classification model in the form of a tree consisting of *decision nodes* and *leaves*. A leaf specifies a class value. A decision node specifies a *test* over one of the predictive attributes, which is called the attribute *selected* at the node. For each possible outcome of the test, a child node is present. The *weight* of a child is the fraction of cases of the training set reaching the decision node that satisfy the test outcome of the child. The *largest child* is the child node with the largest weight. A case is classified by a decision tree by following the path from the root to a leaf according to the decision node tests evaluated on the case attribute values. The class value in the leaf is the class predicted. For cases with missing value for the attribute selected at a node, several options are available [23]. In experiments, we consider the distribution imputation method of C4.5: all branches of the decision node are followed, and the prediction of a leaf in a branch contributes in proportion to the weight of the branch's child node.

We assume that tree induction consists of two phases: tree growing and tree simplification. The tree constructed by the first phase, which typically follows a divide & conquer top-down pattern, is called the *grown* tree (or, the *unpruned* tree). The tree returned by the second phase is called the *trained tree* (or, the *pruned* tree). Some simplification methods work on the same dataset used for tree growing, namely on the whole training set. Other methods require that the training set is split into two subsets, the *growing set* and the *pruning set*, with tree growing working on the growing set, and tree pruning working on the pruning set or on both. The intended use of the pruning set is to provide unseen cases for evaluating or supporting simplification strategies.

2.2 Datasets The datasets used in experiments and their characteristics are shown in Table 1. With reference to the relation $\mathcal{T}$, we report: the dataset name; the number of cases; the number of discrete, continuous, and the total number of attributes; the number of

Table 1: Datasets used in experiments.

			No. of attributes				Class		Grown Tree (using C4.5 Release 8)		
	$\mathcal{T}$ name	$\|\mathcal{T}\|$	discr.	cont.	total	mis.	no.	base e.r	error rate	tree size	w. depth
1	*Hypo Thyroid*	3,772	22	7	29	8	3	7.71	0.46 ± 0.33	35.4 ± 4.5	2.62 ± 0.22
2	*Blocks*	5,473		10	10		4	10.23	3.24 ± 0.63	126 ± 14	8.09 ± 0.60
3	*Musk Clean2*	6,598		166	166		2	15.41	3.16 ± 0.75	236 ± 10	11.77 ± 0.45
4	*Mushroom*	8,124	22		22	1	2	48.20	0.00 ± 0.00	29.1 ± 0.4	2.53 ± 0.03
5	*Letter*	20,000		16	16		26	95.93	12.03 ± 0.76	2563 ± 35	13.27 ± 0.47
6	*Chess*	28,056	3	3	6		18	83.77	31.52 ± 0.96	9898 ± 53	9.11 ± 0.03
7	*Adult*	48,842	8	6	14	3	2	23.93	15.23 ± 0.45	9496 ± 248	11.04 ± 0.22
8	*Statlog Shuttle*	58,000		9	9		7	21.40	0.02 ± 0.02	58.5 ± 3.6	6.96 ± 0.44
9	*Connect-4*	67,557	42		42		3	34.17	20.45 ± 0.44	14735 ± 124	12.37 ± 0.10
10	*SyD100KP5G0.2*	100,000	3	6	9		2	20.00	4.74 ± 0.22	17339 ± 527	8.21 ± 0.11
11	*Census-Income*	299,285	33	7	40	8	2	6.20	5.53 ± 0.10	71925 ± 979	11.46 ± 0.22
12	*Forest Cover*	581,012	44	10	54		7	51.24	5.43 ± 0.10	32706 ± 298	30.48 ± 0.15
13	*SyD1MP7G0.1*	1,000,000	3	6	9		2	10.00	1.68 ± 0.06	65949 ± 1083	8.03 ± 0.10
14	*KDD Cup 99*	4,898,431	7	34	41		23	42.68	0.01 ± 0.00	1980 ± 165	8.12 ± 0.21

attributes with at least one missing value; the number of class values; the *base error rate*, namely the error rate of classifying all cases from the dataset with the most frequent class value. With reference to grown decision trees induced by C4.5 Release 8, we report the tree size, as per number of nodes, the average depth of paths followed in classifying test cases, called the *weighted depth* [19], and the error rate on a repeated 10-fold stratified cross-validation. All datasets are publicly available from the UCI Machine Learning repository [13], apart from *10* and *13* which are synthetically generated using the QUEST data generator [15]. Dataset *10* (resp., *13*) was produced using function 5 (resp., 7) with a class distribution of 20%-80% (resp., 10%-90%).

2.3 Experimental Setup We adhere here to a *repeated stratified* 10-fold cross-validation methodology. 10-fold cross validation has been demonstrated to be a nearly unbiased estimator [17], yet highly variable for small datasets, with Kohavi's final recommendation to adopt a stratified version of it. We ameliorate possible variability of the estimator by adopting repetition of cross validation, as discussed in [16].

All experiments reported in this paper were obtained by repeating 5 times a stratified 10-fold cross-validation. For simplification methods requiring the splitting of the training set, we further adopted a 70%-30% *stratified* splitting into growing set and pruning set. Results are shown in the form $xx \pm yy$ (see Table 1), where xx is the mean value over the 50 executions, and yy is the sample standard deviation. For tree size and weighted depth, xx and yy are absolute values. For error rate, xx and yy are percentage values, e.g., 8.5 ± 0.6 means an average error rate of 8.5% and a sample standard deviation of 0.6%. When comparing two simplification methods (e.g., pruning vs grafting), we adopt the standard approach of [7], which resorts to a two-tailed paired t-test of the differences of error rates over the 50 executions at some significance level.

For tree growing, we adopted a C++ implementation of C4.5 Release 8 [22]. The parametric simplification algorithm proposed in this paper was developed on top of it, with all source code written in standard C++. Elapsed times reported in the paper consider only the tree simplification phase, not tree growing. They refer to a common PC desktop architecture: Xeon 5150 @2.66GHz 4MB L2 cache and 2 GBytes of main memory, with Linux x86_64 2.6.18.

3 Simplification by Pruning and Grafting

3.1 Data Structures and Error Accumulation

Before entering the details of the pruning and grafting algorithms, let us introduce the data structures they work on: cases, nodes, and trees. Here, and for the rest of the paper, we follow a (as much as reasonable) C++ syntax, with the intent of conveying a clear and unambiguous description of the proposed algorithms.

```
class acase {
  ...
 int classValue;
 double weight;
};

typedef vector<acase> &cases;

class tree {
  ...
 node *root;
};

class node {
  ...
  size_t  nChilds;
  size_t  nLeaves;
 int classValue;
 node *get_child(int i);
 node *largest(cases c);
};
```

```
      template<class error_est>
2.2   double node::acc(cases c) {
       if(nChilds == 0)
2.4      return error_est(c);
       double tree_err = 0;
2.6    for(int i=0; i < nChilds; i++) {
          node *child = get_child(i);
2.8       cases cc = select(c, i);
          tree_err += child->acc(cc);
2.10      release(cc);
       }
2.12   return tree_err;
      }

      double node::base_err(cases c) {
2.16   double err = 0;
       for(size_t i=0;i<c.size();++i)
2.18     if(c[i].classValue!=classValue)
           err += c[i].weight;
2.20   return err;
      }
```

Figure 2: Accumulation algorithm.

```
      template<class error_est>
3.2   double node::simp(cases c, cases t) {
       classValue = most_freq(t);
3.4    double n_err = error_est(c);
       if(nChilds == 0)
3.6       return n_err;
       double tree_err = 0;
3.8    for(int i=0; i < nChilds; i++) {
         node *child = get_child(i);
3.10     cases cc = select(c, i);
         cases tc = select(t, i);
3.12     tree_err += child->simp(cc, tc);
         release(cc);
3.14     release(tc);
       }
3.16   // grafting fragment start
       node *large = largest(t);
3.18   double l_err =
         large->acc<error_est>(c);
3.20   if( l_err <= tree_err
             && l_err < n_err) {
3.22     replace_with_child(large);
         return simp(c, t);
3.24   }
       // grafting fragment end
3.26   if(n_err <= tree_err) {
         make_a_leaf();
3.28     return n_err;
       }
3.30   return tree_err;
      }

      void tree::EBPsimp() {
3.34    cases c = training_cases();
        root->simp<ebp_est>(c, c);
3.36  }

3.38  void tree::REPsimp() {
        cases c = pruning_cases();
3.40    cases t = growing_cases();
        root->simp<base_err>(c, t);
3.42  }

3.44  double tree::MEPsimp() {
        cases c = growing_cases();
3.46    root->simp<mep_est>(c, c);
        cases p = pruning_cases();
3.48    return root->acc<base_err>(p);
      }
```

Figure 3: Bottom-up simplification, and EBP, REP, and MEP instances.

A case contains a reference to the attribute values (not shown here, since it is not relevant for our purposes), the class value, modelled here as an integer data type, and the case weight. Cases can be weighted by the user to set their relevance in the classification task (i.e., weights are a further input of the classification task), or by the distribution imputation method of C4.5 when dealing with missing attribute values. The *cases* data type is a reference to a vector of cases. A tree contains a pointer *root* to the root node. A node stores the number *nChilds* of child nodes (zero in case it is a leaf), the number *nLeaves* of leaves in its subtree, and the most frequent class *classValue* of cases from the training set reaching the node (for leaf nodes, this is the predicted class value). Also, references to child nodes are stored (not shown), and accessed by the *get_child* method. Finally, the *largest(c)* method returns the child with the largest fraction of cases in *c* that satisfy the test outcome of the child. It is worth noting that the largest child cannot be fixed statically, due to changes in the structure of the tree as an effect of grafting nodes.

EXAMPLE 1. Consider the sample tree from Fig. 1. Assume that there are 3 cases of the training set reaching node 3, with 2 having $a_3 = t$ and 1 having $a_3 = f$. Hence node 5 is the largest child of node 3. After grafting node 3 (right-hand side tree), cases previously at the sibling 4 reach now node 3. Assume there are 2 of them, both with $a_3 = f$. As a result, node 6 becomes now the largest child of node 3.

Fig. 2 reports a simple parametric method *node::acc* for accumulating the result of an error estimation function *error_est* evaluated on the leaves of a subtree starting from a set of cases *c*. Since the pruning and grafting algorithms that we will consider follow a similar bottom-up pattern, let us discuss it in more detail. For a leaf node, it simply returns the result of *error_est* over cases in *c* (§2.4 — throughout the paper, we use the §M.N to reference line N from the pseudo-code in Fig. M). Otherwise, it accumulates in *tree_err* the results of recursive calls over the child nodes (§2.9), and returns it as the final result (§2.12).

EXAMPLE 2. Consider the *node::base_err* method reported in Fig. 2 (§2.15-21). It calculates the weighted number of misclassified cases at a leaf node. The call *root—>acc<base_err>(c)* returns the weighted number of cases in *c* misclassified by the decision tree rooted at the node pointer *root*.

Finally, we point out the attention on the call *select(c,i)* at (§2.8). It computes the subset of cases in *c* that follow the branch towards the i^{th} child. The call *release(cc)* at (§2.10) performs some restoring operations, depending on the actual implementation of *select*, as described next.

EXAMPLE 3. The C4.5 [20] and `dti` [3] systems implement *select* by rearranging the cases that follow the branch towards a child at the beginning of the vector storing cases. In the distribution imputation method, cases with missing value of the attribute selected at the node are passed to each child with a scaled weight. The role of *release* is then to reset the original weights. The YaDT [22] and Weka [27] systems implement *select* by allocating a new vector of cases. The method *release* now reduces to its deallocation. This is more memory consuming than C4.5, but more efficient and it allows for a direct parallelisation [1].

3.2 Parametric Algorithm EBP, REP and MEP share a common bottom-up algorithm, shown in Fig. 3, which is parametric in the error estimation function *error_est*. The *node::simp* method visits the subtree of a node starting from a pair of sets of cases c and t. c is used to calculate the error estimates of the various alternatives (pruning, grafting, or none). t is the subset of the training set reaching the node, and it is used to update the most frequent class at the node. *node::simp* possibly prunes decision nodes or grafts largest childs, and it returns the error estimate accumulated on the resulting subtree. Let us present the algorithm in detail.

For the current node, the error estimate *n_err* of turning the node into a leaf is calculated first (§3.4). If the node is actually a leaf, such an estimate is simply returned (§3.6). Otherwise, the error estimates of all child subtrees are accumulated in the *tree_err* variable through recursive calls (§3.12), which in turn may prune and/or graft subtrees.

Do not consider for a while the grafting fragment delimited by the C++ comments (§3.16-25). If the error estimate *n_err* of the node as if it would be a leaf is lower or equal than the error estimate *tree_err* of the whole subtree, then the node is turned into a leaf (§3.27). This is the basic condition of decision tree pruning. The return value is *n_err* or *tree_err* on the basis of the final state of the subtree.

Consider now the grafting fragment. The additional error estimate *l_err* is computed on the largest child node *for the cases* c *at the current node* (§3.17-19) by accumulating the error estimates of the leaves from the largest child subtree[2]. Intuitively, this means considering the error estimate of the subtree obtained by grafting the largest child. The basic pruning condition naturally extends to grafting by testing whether *l_err* is lower than *n_err* (i.e., grafting is better than pruning) and lower or equal than *tree_err* (i.e., grafting is not worse than leaving the subtree as it is). If this is the case, the current subtree is replaced by its largest child subtree by calling the method *node::replace_with_child* at (§3.22). In such a case, we say that *subtree replacement takes place*. In addition, the simplification procedure is repeated on the new current node (the previous largest child) (§3.23), due to two reasons. First, since cases c now distribute differently in the grafted subtree, additional pruning and/or grafting may take place. Second, since cases t now distribute differently in the grafted subtree, the most frequent class value at a node (and, hence, the class predicted at a leaf) must be updated. This second task is performed during the recursive visit at (§3.3). Strictly speaking, this is not needed in pruning without grafting, since pruning alone does not affect the distribution of cases.

An alternative option to grafting the largest child consists of considering every child as a candidate for grafting. This is easily implemented by computing the minimum value of *l_err* at (§3.18-19) over all child nodes, and, if grafting takes place, by calling *replace_with_child* over the child that yielded that minimum value (if there is more than one, choose the child with the smallest subtree).

3.3 Algorithm Instances: EBP, REP, and MEP Simplification The parametric algorithm in Fig. 3 is instantiated by specifying the error estimation function *error_est* and the initial set of cases c, while t is fixed to the set of cases used to grow the tree (the whole training set or the growing set depending on the error estimation function). Let us see how to derive the EBP, REP, and MEP simplification methods – while referring the reader to survey papers [6, 11, 21] for details on their error estimation functions.

EBP adopts the C4.5 pessimistic error estimate *ebp_est* (not shown) based on binomial distribution confidence limits at some confidence level. In experiments, we leave the C4.5 default confidence level of 25%. The pessimistic error estimate works on cases from the training set.

REP adopts the simple error count estimation, i.e., the *base_err* method from Fig. 2. It works on the pruning set.

MEP adopts the expected error rate estimate[3] *mep_est* (not shown) based on the m-probability of a class value. MEP was originally proposed to work on the training set, but the m parameter has no default value. For this reason, we adhere to [11] by splitting the training set into growing set and pruning set. In experiments, MEP will be called on the growing set for a collection of m values[4]. The pruned tree with the lowest error rate over the pruning set is chosen as the final pruned tree.

[2]Notice that such an error estimate cannot be computed during the recursive call (§3.12), since it refers to the cases c at the current node, not to the cases cc at the largest child.

[3]Since the expected error rate of a subtree is defined as the *weighted* sum of the expected error rates of its child subtrees, the accumulation (§3.12) has to be changed accordingly. An alternative way, which fully adheres to the accumulation schema of *node::simp*, consists of coding *mep_est* to return the expected error rate multiplied by the number of cases at the node.

[4]$m \in \{0.5, 1, 2, 3, 4, 8, 12, 16, 32, 64, 128, 512, 1024\}$.

```
     void node::simpTD(cases c) {
4.2   classValue = most_freq(c);
      double n_err = base_err(c)+0.5;
4.4   if(nChilds == 0)
          return n_err;
4.6   double tree_err = correct(c, acc<base_err>(c));
      // grafting fragment start
4.8   node *large = largest(c);
      double l_err = correct(c, large->acc<base_err>(c));
4.10  if( l_err <= tree_err && l_err < n_err) {
         replace_with_child (large);
4.12     simpTD(c);
         return;
4.14  }
      // grafting fragment end
4.16  if(n_err <= tree_err) {
         make_a_leaf();
4.18     return;
      }
4.20  for(int i=0; i < nChilds; i++) {
         node *child = get_child(i);
4.22     cases cc = select(c, i);
         child->simpTD(cc);
4.24     release(cc);
      }
4.26 }

4.28 double node::correct(cases c, double err) {
      err += nLeaves/2;
4.30  double nc = c.size();
      err += sqrt(err*(nc-err)/nc);
4.32  return err;
     }

     void tree::PEPsimp() {
4.36   cases c = training_cases();
       root->simpTD(c);
4.38 }
```

Figure 4: PEP simplification algorithm.

We code these instances respectively by the methods *EBPsimp*, *REPsimp*, and *MEPsimp* shown in Fig. 3. Notice that for EBP and MEP, the parameters c and t refer to the same set of cases, hence the actual implementation can benefit of this, e.g., (§3.11,§3.14) can be omitted.

To take the notation simple, for an error estimation function X, we denote by X-p the instance of the parametric algorithm in Fig. 3 without the grafting fragment, by X-g the instance with the grafting fragment, and by X-a the instance (not shown in the figure) with the option of considering every child as candidate for grafting. As an example, EBP-p denotes error based pruning *without* grafting, EBP-g pruning *with* grafting of the *largest* child, and EBP-a pruning *with* grafting of *any* child. In particular, EBP-g is the original simplification algorithm of the C4.5 system.

3.4 A Top-Down Variant: PEP Simplification

The integration of grafting and PEP pruning consists of a *top-down* variant of the algorithm in Fig. 3. Top-down means that the for-loop over child nodes is moved after the pruning and grafting tests, as shown in Fig. 4. Since

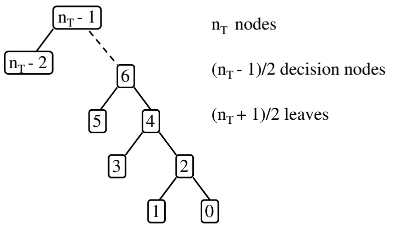

Figure 5: A sample binary decision tree.

the pessimistic error estimation function works on the training cases, the *node::simpTD* method is reported with only the parameter c. The PEP error estimate for a leaf is the corrected base error (§4.3). For a decision node, it is a pessimistic estimate of the corrected subtree error, calculated by the *correct* function in (§4.28-33). Notice that simplification acts top-down, but it has to be performed *after* the tree is grown, since the whole subtree of a node is required for calculating the error estimates *tree_err*[5] (§4.6) and the error estimate of grafting the largest child *l_err* (§4.9).

4 Complexity, Efficiency, Effectiveness

4.1 Complexity While pruning is universally recognized as an efficient simplification method, requiring only a bottom-up tree transversal, grafting is deemed a complex method, yet its complexity has not been thoroughly investigated. Let us shed some light on the computational burden of adding grafting to decision tree pruning. We will consider EBP-p, EBP-g, and EBP-a, but the same conclusions apply to the REP, MEP, and PEP instances, since they all share the same algorithmic schema. First of all, we recall some standard definitions and notation on trees. For a tree T, we denote by n_T the number of nodes in T, and by m_T the number of decision nodes (or internal nodes) in T. The depth of a node is the length of the path from the root to the node (the root has depth 0). The *total path length* of T, denoted by $TPL(T)$, is the sum of the depths of all nodes in T [8]. The *average depth* of T is $\delta(T) = TPL(T)/n_T$.

Let us denote by $V_p(T)$, $V_g(T)$, and $V_a(T)$ the number of nodes respectively visited by EBP-p, EBP-g, and EBP-a. The next result provides us with precise characterizations of $V_p(T)$ and of the ratio $V_a(T)/V_p(T)$, and with lower and upper bounds for $V_g(T)/V_p(T)$. The

[5]Actually, the call *acc<base_err>(c)* at (§4.6) is executed only for the root node and when grafting takes place. The values computed for each node in the subtree are cached in a purposely dedicated node member.

Table 2: Elapsed time average ratios.

	EBP-g/ EBP-p	EBP-a/ EBP-g	EBP-a time (s)	REP-g/ REP-p	REP-a/ REP-g	REP-a time (s)	MEP-g/ MEP-p	MEP-a/ MEP-g	MEP-a time (s)	PEP-g/ PEP-p	PEP-a/ PEP-g	PEP-a time (s)
1 *Hypo Thyroid*	1.62	1.89	0.0018	1.88	1.84	0.0017	1.23	1.36	0.0262	2.00	2.14	0.0017
2 *Blocks*	2.31	1.53	0.0077	2.35	1.40	0.0060	1.33	1.21	0.1548	3.23	1.53	0.0088
3 *Musk Clean2*	3.89	1.36	0.0240	3.77	1.33	0.0180	1.99	1.25	0.2824	4.20	1.38	0.0233
4 *Mushroom*	1.44	1.19	0.0028	1.54	1.36	0.0031	1.21	1.17	0.0319	1.73	1.25	0.0022
5 *Letter*	4.59	1.45	0.1426	4.69	1.44	0.1199	2.95	1.39	1.0559	5.34	1.46	0.1448
6 *Chess*	2.81	2.07	0.2339	2.92	2.36	0.2189	2.05	1.81	1.5678	3.49	2.20	0.2368
7 *Adult*	2.53	1.54	0.2218	3.95	1.65	0.3704	3.68	1.52	3.7600	42.09	0.47	0.2876
8 *Statlog Shuttle*	2.19	1.67	0.0725	2.45	1.51	0.0563	1.52	1.53	0.6196	2.57	1.48	0.0698
9 *Connect-4*	3.57	1.57	0.4333	4.12	1.60	0.4554	3.59	1.52	5.0027	6.21	1.60	0.5665
10 *SyD100KP5G0.2*	1.54	1.67	0.2574	1.91	2.28	0.3975	1.53	1.84	3.3860	14.01	2.06	0.6373
11 *Census-Income*	1.64	1.73	1.1612	3.56	2.07	3.1486	3.24	2.07	43.5307	13.36	7.60	2.5818
12 *Forest Cover*	9.25	1.34	16.7775	10.61	1.34	16.6491	6.20	1.33	134.4037	9.77	1.34	17.0824
13 *SyD1MP7G0.1*	2.21	2.55	8.4421	3.08	2.50	9.8591	2.31	2.80	87.1998	7.24	3.42	14.5895
14 *KDD Cup 99*	3.31	5.56	62.7238	3.76	4.43	50.1524	1.45	3.41	510.3884	7.69	3.79	73.3375

result can be read as follows:

- $V_p(T) = n_T$: pruning alone consists of a linear visit of the decision tree, as one would expect;
- $V_a(T)/V_p(T) = 1 + \delta(T)$: grafting any child has exactly $1 + \delta(T)$ times the complexity of pruning alone, where $\delta(T)$ is the average depth of the decision tree;
- the complexity of grafting the largest child is in between 1 and $1 + \delta(T)$ times the complexity of pruning alone.

The second and third characterizations hold under the assumption that no subtree replacement takes place, i.e., that the test at (§3.20-21) is always false.

THEOREM 4.1. *For a decision tree T, we have $V_p(T) = n_T$. Moreover, under the assumption that no subtree replacement takes place, we have:*

$$\begin{aligned}\frac{k+1}{k} - \frac{1}{kn_T} &\leq \frac{V_g(T)}{V_p(T)} \\ &\leq \frac{k-1}{k} + \frac{1}{kn_T} + \delta(T) \\ &\leq 1 + \delta(T) = \frac{V_a(T)}{V_p(T)}\end{aligned}$$

where $k \geq 2$ is the maximum out-degree of a node in T.

Proof. EBP-p visits each node exactly once, thus $V_p(T) = n_T$. The number of nodes visited by EBP-a is $V_a(T) = n_T + \sum_{sub(T')} n_{T'}$, where $sub(T')$ holds iff T' is a subtree of T not equal to T itself. In fact, n_T nodes are visited bottom-up, while whole trees T', such that $sub(T')$ holds, are visited to compute the error estimates of grafting any child. Since each node contributes to the second addend a number of times equal to its depth, we have $V_a(T) = n_T + TPL(T)$. By definition of $\delta(T)$ and since $V_p(T) = n_T$, we conclude the rightmost equality $V_a(T)/V_p(T) = 1 + \delta(T)$.

The number of nodes visited by EBP-g is: $V_g(T) = n_T + \sum_{largest(T')} n_{T'}$, where *largest*$(T')$ holds iff T' is a subtree of T whose top node is the largest child of its father. This implies $V_g(T) \geq n_T + \sum_{largest(T')} 1$. The cardinality of *largest* is equal to the number m_T of decision nodes in T, since every decision node has a largest child. It is well-known that for trees of degree at most k, it turns out that $n_T \leq km_T + 1$. The inequality $V_g(T) \geq n_T + (n_T - 1)/k = (k+1)n_T/k - 1/k$ directly follows. Dividing by n_T yields the lower bound.

The difference $V_a(T) - V_g(T)$ consists of the sum of nodes from non-largest child subtrees. This is minimum when there is only one non-largest child for each decision node (recall that $k \geq 2$), namely $V_a(T) - V_g(T) \geq m_T$ holds. This and $n_T \leq km_T + 1$ imply $V_a(T) - V_g(T) \geq (n_T - 1)/k$. Dividing by n_T, we have:

$$V_a(T)/V_p(T) - 1/k + 1/(kn_T) \geq V_g(T)/V_p(T).$$

Since $V_a(T)/V_p(T) = 1 + \delta(T)$ has been already shown, we conclude the upper bound $\delta(T) + (k-1)/k + 1/(kn_T) \geq V_g(T)/V_p(T)$. Finally, the inequality $(k-1)/k + 1/(kn_T) + \delta(T) \leq 1 + \delta(T)$ trivially holds for $n_T \geq 1$. $\square$

The lower and upper bounds on $V_g(T)/V_p(T)$ are tight. They can be reached as shown in the next two examples. The third example shows that, under the assumption that no subtree replacement takes place, EBP-g and EBP-a are in the worst case quadratic in n_T. In all of the three examples, we refer to the binary tree in Fig. 5.

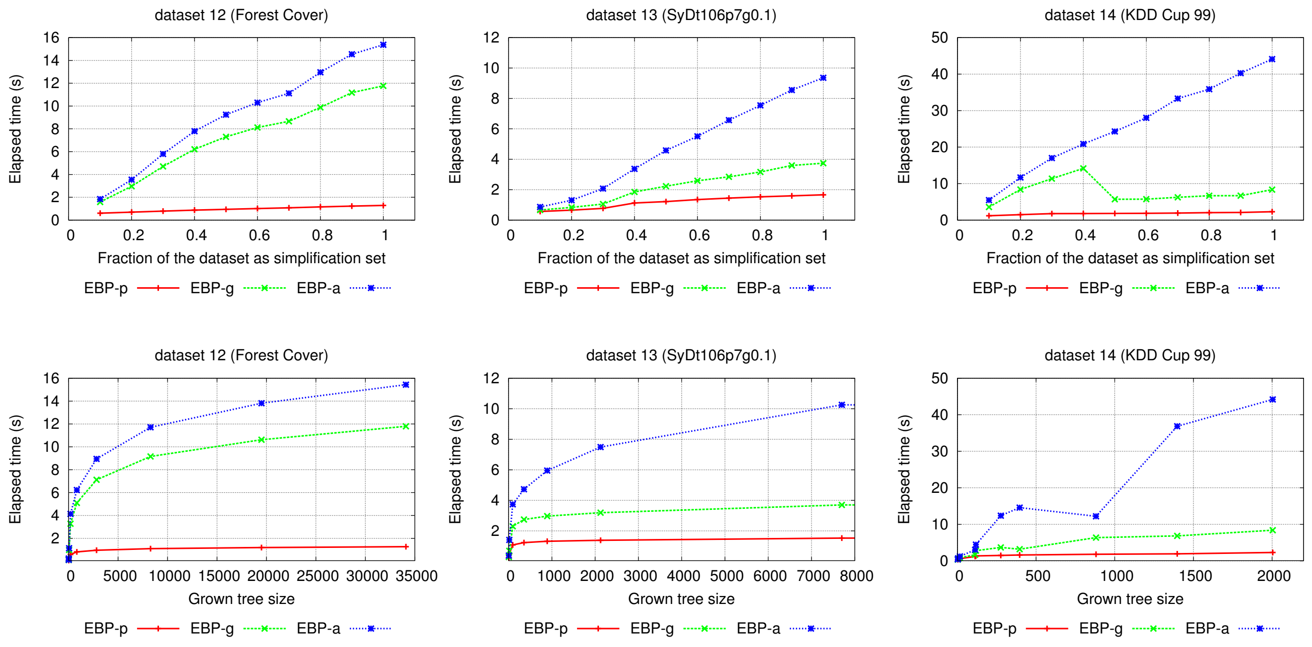

Figure 6: Top: elapsed time over simplification set size as a fraction of the dataset for a fixed grown tree (the one grown on the whole dataset). Bottom: elapsed time over grown tree size for the simplification set fixed to the whole dataset.

Example 4. Assume that the largest child of a decision node is always its left child. We have that $V_g(T) = n_T + (n_T - 1)/2$, since n_T nodes are visited bottom-up, and $(n_T - 1)/2$ nodes are visited as largest childs. Thus, $V_g(T) = (3n_T - 1)/2$ and then $V_g(T)/V_p(T) = 3/2 - 1/(2n_T)$, which is precisely the lower bound of Thm. 4.1 for $k = 2$.

Example 5. Assume now that the largest child is always the right child. By observing that there are exactly 2 nodes at depth d, for $d = 1 \ldots (n_T - 1)/2$, it turns out $TPL(T) = \sum_{i=1}^{(n_T-1)/2} 2i$. We have then $V_g(T) = n_T + \sum_{i=1}^{(n_T-1)/2}(2i - 1)$, since n_T nodes are visited bottom-up, and for the decision node $2i$, with $i = 1 \ldots (n_T - 1)/2$, there are $2i - 1$ nodes in its largest child's subtree. By basic algebra, $V_g(T) = (n_T + 1)/2 + \sum_{i=1}^{(n_T-1)/2} 2i = (n_T + 1)/2 + TPL(T)$, and then $V_g(T)/V_p(T) = 1/2 + 1/(2n_T) + \delta(T)$, which is precisely the upper bound of Thm. 4.1 for $k = 2$.

Example 6. By Thm. 4.1, since $\delta(T) \leq n_T$ clearly holds, $V_g(T)$ and $V_a(T)$ are at worst quadratic in n_T. Let us show that such a bound can be reached for the tree in Fig. 5. Assume again that the largest child of a decision node is always its right child. Since $TPL(T) = \sum_{i=1}^{(n_T-1)/2} 2i = (n_T^2 - 1)/2$, we have $\delta(T) = n_T/2 - 1/(2n_T)$. By Thm. 4.1 and the previous example, both $V_g(T)$ and $V_a(T)$ are then quadratic in n_T.

4.2 Efficiency

Table 2 reports the actual ratios of the elapsed times of X-g over X-p, and of X-a over X-g for X being EBP, REP, MEP, and PEP over the 14 experimental datasets. Also, the absolute elapsed times of the X-a procedures are shown. The values in the table are averaged over 50 executions (10 fold cross-validation repeated 5 times).

The ratios EBP-g/EBP-p, REP-g/REP-p, and MEP-g/MEP-p remain below 5 for all datasets, apart for dataset *12*. The ratio PEP-g/PEP-p remains below 14, with the exception of dataset *7*.

The ratios EBP-a/EBP-g, REP-a/REP-g, and MEP-a/MEP-g are below 3 apart for dataset *14*. The ratio PEP-a/PEP-g remains below 4, apart for *11*.

In summary, adding grafting the largest child to pruning does not increase in the order of magnitude of the elapsed time of pruning for EBP, REP, and MEP, while it is an order of magnitude higher for PEP. The further option of grafting any child does not change the order of magnitude of grafting the largest child for any of the four methods.

It should be also noticed that, in absolute terms, the elapsed times reported in Table 2 for the X-a procedures

Table 3: Effectiveness of EBP simplification.

	Error rate			Tree size			Weighted depth		
	EBP-p	EBP-g	EBP-a	EBP-p	EBP-g	EBP-a	EBP-p	EBP-g	EBP-a
1 *Hypo Thyroid*	0.44 ± 0.37	**0.43** ± **0.35**	**0.43** ± **0.35**	30.2 ± 4.0	**29.4** ± **3.4**	**29.4** ± **3.4**	2.55 ± 0.04	**2.53** ± **0.04**	**2.53** ± **0.04**
2 *Blocks*	3.13 ± 0.61	3.12 ± 0.63	**3.11** ± **0.63**	89.7 ± 9.1	85.0 ± 8.3	**84.4** ± **8.5**	7.04 ± 0.22	**7.01** ± **0.22**	7.06 ± 0.26
3 *Musk Clean2*	3.16 ± 0.76	**3.14** ± **0.77**	**3.14** ± **0.77**	226 ± 9.6	221 ± 10	**219** ± **9.8**	11.67 ± 0.41	11.59 ± 0.43	**11.58** ± **0.44**
4 *Mushroom*	**0.00** ± **0.00**	**0.00** ± **0.00**	**0.00** ± **0.00**	29.1 ± 0.4	29.1 ± 0.4	**29.1** ± **0.2**	2.53 ± 0.03	2.53 ± 0.03	**2.53** ± **0.02**
5 *Letter*	12.06 ± 0.78	**12.03** ± **0.79**	**12.03** ± **0.79**	2394 ± 43	2340 ± 37	**2338** ± **36**	12.99 ± 0.50	**12.86** ± **0.44**	12.87 ± 0.44
6 *Chess*	31.84 ± 0.89	31.28 ± 0.87	**31.16** ± **0.85**	8644 ± 79	8278 ± 79	**8165** ± **75**	8.88 ± 0.03	**8.86** ± **0.03**	**8.86** ± **0.03**
7 *Adult*	13.90 ± 0.42	13.92 ± 0.43	**13.86** ± **0.43**	1013 ± 130	718 ± 108	**679** ± **103**	6.80 ± 0.16	**6.69** ± **0.21**	6.79 ± 0.23
8 *Statlog Shuttle*	**0.02** ± **0.02**	0.03 ± 0.02	0.03 ± 0.02	55.0 ± 3.4	47.4 ± 3.3	**47.2** ± **2.9**	6.89 ± 0.45	**5.79** ± **0.29**	5.86 ± 0.29
9 *Connect-4*	19.16 ± 0.45	18.98 ± 0.42	**18.96** ± **0.44**	6442 ± 142	6038 ± 118	**5989** ± **116**	10.12 ± 0.09	**9.98** ± **0.09**	**9.98** ± **0.09**
10 *SyD100KP5G0.2*	3.72 ± 0.18	3.68 ± 0.18	**3.62** ± **0.18**	**274** ± **17**	286 ± 16	296 ± 13	**6.26** ± **0.21**	6.33 ± 0.20	6.38 ± 0.20
11 *Census-Income*	4.69 ± 0.09	**4.59** ± **0.09**	**4.59** ± **0.09**	2805 ± 581	2559 ± 292	**2451** ± **261**	**5.52** ± **0.09**	5.56 ± 0.13	5.59 ± 0.13
12 *Forest Cover*	5.43 ± 0.10	**5.41** ± **0.11**	**5.41** ± **0.11**	29563 ± 240	28530 ± 240	**28422** ± **232**	30.12 ± 0.14	**29.88** ± **0.14**	29.89 ± 0.15
13 *SyD1MP7G0.1*	1.40 ± 0.05	1.38 ± 0.05	**1.37** ± **0.05**	2820 ± 143	2821 ± 128	**2785** ± **126**	**6.86** ± **0.06**	6.87 ± 0.06	6.90 ± 0.06
14 *KDD Cup 99*	**0.01** ± **0.00**	**0.01** ± **0.00**	**0.01** ± **0.00**	1234 ± 110	1123 ± 123	**1068** ± **119**	7.15 ± 0.29	**6.91** ± **0.23**	13.61 ± 3.13

remain reasonably low in all cases except for the largest dataset. MEP-a exhibits the highest values, but this is well-explained by recalling that at each of the 50 executions the MEP-a procedure is actually called 13 times to select the best parameter m (see footnote 4).

4.3 Scalability Let us consider how the simplification procedures scale with respect to the size of their two inputs considered in isolation: the simplification set and the grown tree to be simplified. By simplification set, we denote the union of cases c and t in input to *node::simp*, namely the training set for EBP, REP, and PEP, and the growing set for MEP.

Simplification set. Since *select* and the error estimation functions of EBP, REP, MEP, and PEP are linear in the number of cases, we expect that, for a fixed tree, the elapsed time of simplification algorithms is approximatively linear in the size of the simplification set. This certainly holds for pruning in isolation, since it is a pure bottom-up tree traversal of the grown tree. In the case of grafting, this holds only if the number of pruning and/or subtree replacement operations do not vary with the (size of the) simplification set. Fig. 6 (top) shows the elapsed times of EBP-p, EBP-g, and EBP-a for a same decision tree (built from the whole dataset) for different fractions of the dataset as simplification set, over the three largest datasets (*12*, *13*, and *14*). Linearity is apparent across the datasets and the simplification algorithms. A case worth noting is EBP-g for dataset *14*, which has a reduction in elapsed time from fraction 40% to 50% of the dataset. This is due to the fact that a child of the root node is grafted up to fraction 40%, but not after. Since the grafted subtree is revisited, such an additional, considerable, cost stops at fraction 50%.

Grown tree size. Thm. 4.1 provides us with theoretical bounds on the number of nodes visited by the simplification procedures. However, the cost of visiting a node is not constant, but, as already mentioned, linear in the number of cases. Since the cases at a decision node are partitioned among the child nodes, we can expect that the elapsed time of the visit performed by X-p grows sub-linearly in the tree size (logarithmically for balanced trees). For X-g and X-a, the same reasoning leads to a sub-quadratic conclusion (quadratic in the logarithm of the tree size for balanced trees). The plots in Fig. 6 (bottom) report the elapsed times of EBP-p, EBP-g, and EBP-a for trees of increasing size over the three largest datasets. Technically, decision trees of increasing size but with the same top-level structure are obtained by setting the C4.5 option of stopping-earlier when the number of cases at a node reaches a minimum threshold. In all plots, the simplification set is fixed to the whole dataset, so that, as in the previous discussion, we are testing the impact of varying only one input of the simplification procedures. It is readily checked from Fig. 6 (bottom) that the elapsed times grow sub-linearly with the size of the tree, with the top-level nodes taking most of the elapsed time.

4.4 Effectiveness A simplification method is effective if it reduces tree size, weighted depth, or other structural metrics of decision trees while not worsening accuracy. Even in the case of equal accuracy, a method that is able to reduce tree complexity is worth being adopted:

- in descriptive tasks, since it provides a more compact description of the class values;
- and in predictive tasks, since it allows for a faster prediction time.

Table 3 reports the details of error rate, tree size, and weighted depth for EBP-p, EBP-g, and EBP-a over the 50 experimental executions.

Concerning error rates, grafting is at least as accurate as pruning and sometimes is more accurate, with a

Table 4: Paired t-test for statistical significance of the differences.

			EBP-p - EBP-g	EBP-g - EBP-a	REP-p - REP-g	REP-g - REP-a	MEP-p - MEP-g	MEP-g - MEP-a	PEP-p - PEP-g	PEP-g - PEP-a
Error Rate	1	*Hypo Thyroid*								
	2	*Blocks*								
	3	*Musk Clean2*						−(0.99)	+(1.00)	
	4	*Mushroom*								
	5	*Letter*					+(0.99)	−(0.97)	+(1.00)	
	6	*Chess*	+(1.00)	+(1.00)			+(1.00)	+(1.00)	+(1.00)	+(1.00)
	7	*Adult*		+(1.00)				−(1.00)	+(1.00)	−(1.00)
	8	*Statlog Shuttle*	−(1.00)	+(0.98)			−(1.00)	+(1.00)		+(0.99)
	9	*Connect-4*	+(1.00)	+(0.99)	+(0.99)		+(1.00)		+(1.00)	+(1.00)
	10	*SyD100KP5G0.2*	+(1.00)	+(1.00)		+(0.98)		+(1.00)	−(1.00)	+(1.00)
	11	*Census-Income*	+(1.00)					+(0.98)		+(1.00)
	12	*Forest Cover*	+(1.00)		−(0.99)		+(1.00)		+(1.00)	+(1.00)
	13	*SyD1MP7G0.1*	+(1.00)	+(1.00)	+(1.00)		+(1.00)	+(1.00)	+(1.00)	−(1.00)
	14	*KDD Cup 99*	+(0.97)				−(0.99)		+(1.00)	−(0.97)
Tree Size	1	*Hypo Thyroid*	+(1.00)				+(1.00)	+(1.00)		
	2	*Blocks*	+(1.00)	+(0.99)						
	3	*Musk Clean2*	+(1.00)	+(1.00)	+(1.00)		−(0.95)			+(1.00)
	4	*Mushroom*						+(0.99)		
	5	*Letter*	+(1.00)	+(0.98)	+(1.00)	+(1.00)	+(1.00)	+(0.99)	+(1.00)	+(1.00)
	6	*Chess*	+(1.00)	+(1.00)	+(1.00)	+(0.99)	+(1.00)		+(1.00)	+(1.00)
	7	*Adult*	+(1.00)	+(1.00)			+(1.00)		−(1.00)	+(1.00)
	8	*Statlog Shuttle*	+(1.00)		+(1.00)		+(1.00)	−(1.00)	+(1.00)	
	9	*Connect-4*	+(1.00)	+(1.00)	+(1.00)	+(1.00)	+(0.95)	−(0.99)	+(1.00)	+(1.00)
	10	*SyD100KP5G0.2*	−(1.00)	−(1.00)				+(1.00)	−(1.00)	
	11	*Census-Income*	+(1.00)	+(1.00)	+(1.00)		+(1.00)	+(1.00)		−(1.00)
	12	*Forest Cover*	+(1.00)	+(1.00)	+(1.00)	+(1.00)	+(1.00)		+(1.00)	+(1.00)
	13	*SyD1MP7G0.1*	−(1.00)	+(1.00)		+(1.00)	+(1.00)	+(1.00)	−(1.00)	+(1.00)
	14	*KDD Cup 99*	+(1.00)	+(1.00)	+(1.00)	+(1.00)	+(1.00)		−(1.00)	+(1.00)
Weighted Depth	1	*Hypo Thyroid*	+(1.00)				+(0.99)	+(0.96)		
	2	*Blocks*	+(1.00)	−(0.97)				−(1.00)	+(0.99)	
	3	*Musk Clean2*	+(1.00)		+(0.99)				+(0.95)	
	4	*Mushroom*						+(0.99)		
	5	*Letter*	+(1.00)		+(1.00)		+(1.00)	+(0.98)	+(1.00)	
	6	*Chess*	+(1.00)				+(1.00)		−(1.00)	−(1.00)
	7	*Adult*	+(1.00)	−(1.00)	−(0.96)	−(0.99)		−(1.00)	−(1.00)	+(1.00)
	8	*Statlog Shuttle*	+(1.00)	−(1.00)	+(1.00)		+(1.00)	−(1.00)	+(1.00)	−(1.00)
	9	*Connect-4*	+(1.00)		+(1.00)			−(0.99)	+(1.00)	
	10	*SyD100KP5G0.2*	−(1.00)	−(1.00)		−(0.97)	+(0.97)	−(0.98)	−(1.00)	−(1.00)
	11	*Census-Income*	−(0.95)	−(1.00)			+(0.99)			−(1.00)
	12	*Forest Cover*	+(1.00)	−(1.00)	+(1.00)		+(1.00)		+(1.00)	−(1.00)
	13	*SyD1MP7G0.1*	−(1.00)	−(1.00)		−(1.00)	+(1.00)	−(1.00)	−(1.00)	−(1.00)
	14	*KDD Cup 99*	+(1.00)	−(1.00)	+(1.00)	−(1.00)	+(1.00)	−(0.99)	+(1.00)	−(1.00)

lower mean error rate and/or a lower sample standard deviation. EBP-a performs slightly better than EBP-g. This justifies the C4.5 choice of adopting grafting, but it would suggest that the option of grafting any child should be preferred.

Concerning tree size, EBP-a is the best method in almost all cases. Dataset *10* is a paradigmatic example showing that grafting does not necessarily lead to smaller trees than pruning alone. Intuitively, grafting the grandchild of a node n may prevent the subsequent pruning of n to a leaf, since the error estimate for the subtree of n is affected by the change in its structure.

Finally, EBP-g is the champion method with respect to weighted depth. By definition, the largest child carries the largest fraction of the cases at the parent node. Therefore, grafting the largest child, instead of any other child, results in a higher impact on the average depth. Datasets *10*, *11*, and *13* show that grafting does not necessarily reduces the weighted depth compared to pruning alone.

Table 4 shows the results of a two-tailed paired t-test over the differences in error rate, tree size, and weighted depth for the four simplification methods. Grafting the largest child vs pruning, and grafting any child vs grafting the largest child are compared. A cell in a column labelled "X - Y" contains "+" if the difference of the mean true values of X and Y is positive at a confidence level of 95%, and hence Y performs better than X. The actual confidence level at which it holds is reported in parenthesis, with 1.00 denoting a level higher than 99.5%. Analogously, a cell reports "−" if the difference of the means is negative, and hence Y performs worse than X. Finally, a cell is left blank if neither conclusion can be drawn. In summary, adding grafting to pruning does not yield less accurate decision trees for all of the four methods EBP-g, REP-g, MEP-g, and PEP-g. For the EBP-g and PEP-g simplification methods, it (slightly but statistically significantly) improves error rate for most of the datasets, and grafting any child is even superior to grafting the largest child. Concerning tree size, apart from the paradigmatic dataset *10*, grafting improves over pruning, and, in turn, grafting any child improves over grafting the largest child. Finally, for weighted depth, grafting the largest child is the preferable method.

5 Related Work

To the best of our knowledge, the systematic extension of pruning methods to include grafting has not been addressed in the literature. Also, a comparison of complexity, efficiency, and effectiveness of grafting vs pruning in decision tree simplification has not been considered. As already observed, C4.5 [20] firstly added the grafting operator to error based pruning, thus obtaining, in our notation, EBP-g. Strangely enough, however, C4.5 does not offer the possibility not to perform grafting, i.e., it does not implement EBP-p.

A theoretical framework for the simplification of decision trees, which include both pruning and grafting operators, has been proposed in [12]. Ceci et al. [9] discuss a form of grafting any child with reference to reduced error pruning. Under the view that the bottom-up algorithm is a greedy search in the space of possible grafting operations, they prove that the algorithm returns the smallest tree among those with the lowest error. Strictly speaking, their algorithm differs from REP-a in two aspects. First, only grafting is performed, without pruning. Second, the error estimate of grafting a child is calculated by recursively calling the simplification procedure. Concretely, line (§3.19) becomes *large—>simp(c)*. This has two consequences. First, once subtree replacement takes place, the procedure does not need to be recursively called on the grafted subtree – i.e., (§3.23) simply returns *l_err*. This is because the grafted subtree has been already recursively visited for the cases at the parent. Second, the complexity of the algorithm becomes now exponential.

Example 7. Let us denote by $V(T)$ the number of nodes visited by Ceci et al.' procedure on a tree T. Consider the tree from Fig. 5. We have $V(T) = 3 + 2V(T_r)$, where T_r is the right subtree of T. In fact, the root node is visited once, the left child is visited twice (once for the cases at the node and once for the cases at the parent), and then the procedure is recursively called on the right child twice again (once for the cases at the node and once for the cases at the parent). In terms of the tree size, this means $V(n_T) = 3+2V(n_T-2)$, hence $V(T)$ is in the order of 2^{n_T}.

Finally, our approach can be readily applied to integrate grafting with other old and new bottom-up pruning algorithms, such as CART cost complexity [5], minimum description length [18], cost sensitive [25], and k-norm [28] pruning.

6 Conclusions

With the advent of optimized (sequential and parallel) algorithms for building decision trees, the post-processing phase of tree simplification becomes worth investigating in terms of complexity, efficiency, and effectiveness. One of the less understood simplification operators is the one of grafting, or subtree replacement, originally implemented by C4.5. In this paper, we have introduced a general bottom-up algorithm, parametric in an error estimation function, for simplifying decision trees by integrating pruning and grafting. The algorithm has been instantiated and analyzed over the error based, reduced error, minimum error, and pessimistic error estimation functions. A theoretical analysis of the complexity of the algorithm in terms of number of nodes visited has been presented, with tight lower and upper bounds. From the experimental side, we have drawn the following conclusions on efficiency:

- grafting the largest child is no more than one order of magnitude higher than pruning;
- grafting any child is in the same order of magnitude of grafting the largest child;
- and, in absolute terms, both remain affordable for moderately large datasets.

A statistically-validated evaluation of the effectiveness of adding grafting to pruning has led to the conclusions that:

- grafting any child yields the smallest trees compared to pruning alone and to grafting the largest child;
- grafting the largest child yields the smallest weighted depth compared to pruning alone and to grafting the largest child;
- grafting does not degrade tree accuracy compared to pruning alone; rather, for EBP and PEP, it slightly (but significantly) improves in accuracy.

Summarizing, the recommendation for data mining researchers and practitioners is that grafting can be used in decision tree simplification without incurring in accuracy degradation, without requiring orders of magnitude of additional running time, and with a significant reduction of tree size (for grafting any child) and weighted depth (for grafting the largest child).

References

[1] M. Aldinucci, S. Ruggieri, and M. Torquati. Porting decision tree algorithms to multicore using FastFlow. In *Proc. of European Conf. on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2010)*, volume 6321 of *LNCS*, pages 7–23. Springer, 2010.

[2] M. Bohanec and I. Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15(3):223–250, 1994.

[3] C. Borgelt. A decision tree plug-in for DataEngine. In *Proc. 6th European Congress on Intelligent Techniques and Soft Computing (EUFIT 1998)*, volume 2, pages 1299–1303. Verlag Mainz, 1998. `dti` version 3.23 downloaded from http://www.borgelt.net.

[4] J. P. Bradford, C. Kunz, R. Kohavi, C. Brunk, and C. E. Brodley. Pruning decision trees with misclassification costs. In *Proc. of European Conf. on Machine Learning (ECML 1998)*, volume 1398 of *LNCS*, pages 131–136. Springer, 1998.

[5] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth Publishing Company, 1984.

[6] L. A. Breslow and D. W. Aha. Simplifying decision trees: A survey. *The Knowledge Engineering Review*, 12:1–40, 1997.

[7] W. Buntin and T. Niblett. A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8:75–85, 1992.

[8] H. Cameron and D. Wood. Maximal path length of binary trees. *Discrete Applied Mathematics*, 55(1):15–35, 1994.

[9] M. Ceci, A. Appice, and D. Malerba. Simplification methods for model trees with regression and splitting nodes. In *Proc. of Int. Conf. on Machine Learning and Data Mining in Pattern Recognition (MLDM 2003)*, volume 2734 of *LNCS*, pages 20–34. Springer, 2003.

[10] J. Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[11] F. Esposito, D. Malerba, and G. Semeraro. A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(5):476–491, 1997.

[12] F. Esposito, D. Malerba, G. Semeraro, and V. Tamma. The effects of pruning methods on the predictive accuracy of induced decision trees. *Applied Stochastic Models in Business and Industry*, 15:277–299, 1999.

[13] A. Frank and A. Asuncion. UCI machine learning repository, 2011. http://archive.ics.uci.edu/ml.

[14] J. E. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest — A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/4):127–162, 2000.

[15] IBM Almaden. Quest synthetic data generation code, 2003. http://www.almaden.ibm.com/software/quest.

[16] Ji-Hyun Kim. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational Statistics & Data Analysis*, 53(11):3735 – 3745, 2009.

[17] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI 1995)*, pages 1137–1145. Morgan Kaufmann, 1995.

[18] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining (KDD 1995)*, pages 216–221. AAAI Press, 1995.

[19] K.-M. Osei-Bryson. Post-pruning in decision tree induction using multiple performance measures. *Computers & Operational Research*, 34(11):3331–3345, 2007.

[20] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[21] J. R. Quinlan. Simplifying decision trees. *International Journal of Human-Computer Studies*, 51(2):497–510, 1999.

[22] S. Ruggieri. YaDT: Yet another Decision tree Builder. In *Proc. of Intl. Conf. on Tools with Artificial Intelligence (ICTAI 2004)*, pages 260–265. IEEE, 2004. YaDT version 1.2.5 downloaded from http://www.di.unipi.it/∼ruggieri.

[23] M. Saar-Tsechansky and F. Provost. Handling missing values when applying classification models. *Journal of Machine Learning Research*, 8:1625–1657, 2007.

[24] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *Data Minining & Knowledge Discovery*, 3(3):237–261, 1999.

[25] T. Wang, Z. Qin, Z. Jin, and S. Zhang. Handling overfitting in test cost-sensitive decision tree learning by feature selection, smoothing and pruning. *Journal of Systems and Software*, 83(7):1137–1147, 2010.

[26] G. I. Webb. Decision tree grafting from the all tests but one partition. In *Proc. of Int. Joint Conf. on Artificial Intelligence (IJCAI 1999)*, pages 702–707. Morgan Kaufmann, 1999.

[27] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2nd edition, 2005.

[28] M. Zhong, M. Georgiopoulos, and G. C. Anagnostopoulos. A *k*-norm pruning algorithm for decision tree classifiers based on error rate estimation. *Machine Learning*, 71(1):55–88, 2008.