Anytime Induction of Decision Trees: an Iterative Improvement Approach

Saher Esmeir and **Shaul Markovitch**
Computer Science Department
Technion—Israel Institute of Technology, Haifa 32000, Israel
{esaher , shaulm}@cs.technion.ac.il

Abstract

Most existing decision tree inducers are very fast due to their greedy approach. In many real-life applications, however, we are willing to allocate more time to get better decision trees. Our recently introduced LSID3 contract anytime algorithm allows computation speed to be traded for better tree quality. As a contract algorithm, LSID3 must be allocated its resources a priori, which is not always possible. In this work, we present IIDT, a general framework for interruptible induction of decision trees that need not be allocated resources a priori. The core of our proposed framework is an iterative improvement algorithm that repeatedly selects a subtree whose reconstruction is expected to yield the highest marginal utility. The algorithm then rebuilds the subtree with a higher allocation of resources. IIDT can also be configured to receive training examples as they become available, and is thus appropriate for incremental learning tasks. Empirical evaluation with several hard concepts shows that IIDT exhibits good anytime behavior and significantly outperforms greedy inducers when more time is available. A comparison of IIDT to several modern decision tree learners showed it to be superior.

Introduction

Despite the recent progress in developing advanced induction algorithms, such as SVM (Vapnik 1995), *decision trees* are still considered attractive for many real-life applications. This is due in large part to their interpretability (Hastie, Tibshirani, & Friedman 2001). Craven (1996) listed several reasons why the comprehensibility of a classifier is so important. For one thing, comprehensibility makes it possible for humans to validate the induced model and to generate human-readable explanations for the classifier predictions. When classification cost is high, decision trees may be attractive in that they ask only for the values of the features along a single path from the root to a leaf. In terms of accuracy, decision trees were shown to be competitive with other classifiers for several learning tasks.

Smaller consistent decision trees are likely to have better predictive power than their larger counterparts. This is in accordance with Occam's Razor (Blumer *et al.* 1987; Domingos 1999). The problem of finding the smallest consistent tree, however, is known to be NP-complete (Hyafil & Rivest 1976; Murphy & McCraw 1991). That is why most existing decision tree induction algorithms take a greedy approach and use local heuristics for choosing the best splitting attribute. The greedy approach indeed performs quite well for many learning problems and is able to generate decision trees very fast. In some cases, however, when the concept to learn is hard and the user is willing to allocate more time, the existing greedy algorithms are not able to exploit the additional resources to generate a better decision tree.[1]

Recently, we have introduced the LSID3 algorithm that can induce better decision trees given larger allocation of time (Esmeir & Markovitch 2004). The algorithm evaluates a candidate splitting attribute by estimating the size of the smallest consistent tree under it. The estimation is based on sampling the space of consistent trees, where the size of the sample is determined in advance according to the allocated time. LSID3 requires advanced knowledge of the allocated time and does not guarantee any solution if the contract is not honored. Such algorithms are called *contract* anytime algorithm (Russell & Zilberstein 1996).

In many real-life applications, however, the time allocated for the learning phase is not known a priori. In this work we present IIDT[2], a general framework for *interruptible* induction of decision trees, which does not require that time be allocated in advance and can therefore be interrupted whenever necessary. At the center of our proposed framework is an iterative improvement algorithm that repeatedly selects a subtree whose reconstruction is expected to yield the highest marginal utility. The algorithm then rebuilds this subtree with a higher resource allocation. The algorithm can be configured to prune the tree in order to avoid overfitting. It can also be easily modified to handle incremental learning tasks.

LSID3: a Contract TDIDT Algorithm

In this section we give a brief overview of the LSID3 algorithm, which we use as a component of the interruptible framework that will be presented in the next section.

[1] There are applications where the allocated time is not sufficient for building a tree even by a greedy algorithm. In this work we do not deal with such problems.

[2] Interruptible Induction of Decision Trees

Procedure LSID3-CHOOSE-ATTRIBUTE(E, A, r)
  If $r = 0$ **Return** ID3-CHOOSE-ATTRIBUTE(E, A)
  Foreach $a \in A$
    Foreach $v_i \in \text{domain}(a)$*
      $E_i \leftarrow \{e \in E \mid a(e) = v_i\}$
      $min_i \leftarrow \infty$
      Repeat r **times**
        $T \leftarrow$ SID3($E_i, A - \{a\}$)
        $min_i \leftarrow \min(min_i, \text{SIZE}(T))$
    $total_a \leftarrow \sum_{i=1}^{|\text{domain}(a)|} min_i$
  Return a for which $total_a$ is minimal

* When a is numeric the loop is over all possible splitting tests and a is not filtered out when calling SID3.

Figure 1: Attribute selection in LSID3

LSID3 adopts the general top-down induction of decision trees (TDIDT) scheme, and invests more resources for making better split decisions. For every possible candidate split, LSID3 attempts to estimate the size of the resulting subtree were the split to take place, and favors the one with the smallest expected size. The estimation is based on a biased sample of the space of trees rooted at the evaluated attribute. The sample is obtained using a stochastic version of ID3, called SID3. In SID3, the splitting attribute is chosen semi-randomly with a probability proportional to its information gain. LSID3 is a contract algorithm parameterized by r, the sample size. When r is larger, we expect the resulting estimation to be more accurate and LSID3 to improve. Figure 1 lists the attribute selection components in LSID3.

The runtime of LSID3 grows linearly with r. Let m be the number of examples and $n = |A|$ be the number of attributes. Shavlik, Mooney, & Towell (1991) reported for ID3 an empirically based average-case complexity of $O(mn)$. It is easy to see that the complexity of SID3 is similar to that of ID3. LSID3(r) invokes SID3 r times for each candidate split, and thus the runtime of LSID3(r) can be written as

$$\sum_{i=1}^{n} r \cdot i \cdot O(mi) = \sum_{i=1}^{n} O(rmi^2) = O(rmn^3). \quad (1)$$

As a contract algorithm, LSID3 assumes that the contract parameter is known in advance. In many real-life cases, however, either the contract time is unknown or mapping the available resources to r is difficult. To handle such cases, we need to come up with an interruptible anytime algorithm.

Interruptible Induction of Decision Trees

By definition, every interruptible algorithm can serve as a contract algorithm. Russell & Zilberstein (1996) showed that any contract algorithm $\mathcal{A}$ can be converted into an interruptible algorithm $\mathcal{B}$ with a constant penalty. $\mathcal{B}$ is constructed by running $\mathcal{A}$ repeatedly with exponentially increasing time limits. This general approach can be used to convert LSID3 into an interruptible algorithm. LSID3 gets its contract time in terms of r, the sample size. When $r = 0$, LSID3 is defined to be identical to ID3. Therefore, we first call LSID3 with $r = 0$ and then continue with exponentially increasing values of r, starting from $r = 1$.

Procedure IIDT(E, A)
  $T \leftarrow$ GREEDY-BUILD-TREE(E, A)
  While not-interrupted
    $y \leftarrow$ CHOOSE-NODE(T, E, A)
    $t \leftarrow$ subtree of T rooted at $node$
    $r \leftarrow$ NEXT-R($node$)
    $t' \leftarrow$ REBUILD-TREE(EX(y), ATT(y), r)
    If BETTER(t', t) replace t with t'
  Return T

Procedure EX($node$)
  Return $\{e \in E \mid e$ reaches $node\}$

Procedure ATT($node$)
  Return $\{a \in A \mid a \notin$ ancestor of $node\} \cup$
    $\{a \in A \mid a$ is numeric$\}$

Figure 2: The IIDT framework

One problem with the sequencing approach is the exponential growth of the gaps between the times at which an improved result can be obtained. The reason for this problem is the generality of the sequencing approach, which views the contract algorithm as a black box. Thus, in the case of LSID3, at each iteration the entire decision tree is rebuilt. In addition, the minimal value that can be used for τ is the runtime of LSID3($r = 1$). In many cases we would like to stop the algorithm earlier. When we do, the sequencing approach will return the initial greedy tree.

Our interruptible anytime framework, called IIDT, overcomes these problems by iteratively improving the tree rather than trying to rebuild it. We start by describing the IIDT framework and its components. Then, we show how IIDT can be configured to prune the resulting trees and how it can be modified to handle incremental learning tasks.

The IIDT Algorithm

Iterative improvement approaches, which start with an initial solution and repeatedly modify it, have been successfully applied to many AI domains. The key idea behind these techniques is to choose an element of the current suboptimal solution and improve it by local repairs. IIDT adopts the above approach for decision tree learning and iteratively repairs subtrees. It can serve as an interruptible anytime algorithm because, when interrupted, it can immediately return the currently best solution available.

As in LSID3, IIDT exploits additional resources in an attempt to produce better decision trees. The principal difference between the algorithms is that LSID3 uses the available resources to induce a decision tree top-down, where each decision made at a node is final and does not change. IIDT, however, is not allocated resources in advance and uses extra time resources to repeatedly modify split decisions.

IIDT first performs a quick construction of an initial de-

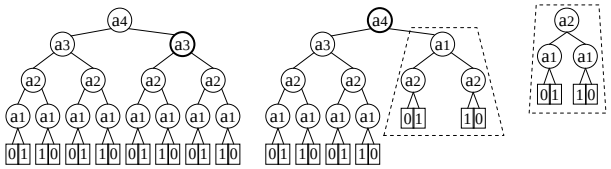

Figure 3: Improving a decision tree by IIDT

Procedure CHOOSE-NODE(T, E, A)
  Foreach $node \in T$
    $r_{node} \leftarrow$ NEXT-R$(node)$
    $cost_{node} \leftarrow$ EXPECTED-COST$(T, node)$
    max-cost $\leftarrow$ EXPECTED-COST$(T, root)$
    If $(cost_{node}/$*max-cost*$) > g$
      $T_{node} \leftarrow$ SUBTREE$(node)$
      $\Delta q \leftarrow$ EXPECTED-BENEFIT(T_{node})
      $u_{node} \leftarrow \Delta q / cost_{node}$
  $best \leftarrow node$ that maximizes u_{node}
  Return $\langle best, r_{best} \rangle$

Figure 4: Choosing a subtree for reconstruction

cision tree by calling ID3. Then it iteratively attempts to improve the current tree by choosing a node and rebuilding its subtree with more resources than those used previously. If the newly induced subtree is better, the existing subtree is replaced. IIDT is formalized in Figure 2. Figure 3 illustrates how IIDT works. The target concept is $a_1 \oplus a_2$, with two additional irrelevant attributes, a_3 and a_4. The leftmost tree was constructed using ID3. In the first iteration, the subtree rooted at the bolded node is selected for improvement and replaced by a smaller tree (surrounded by a dashed line). Next, the root is selected for improvement and the entire tree is replaced by a tree that perfectly describes the concept.

As a general framework, IIDT can use different approaches for choosing which node to improve, for allocating resources, for rebuilding a subtree, and for deciding whether an alternative subtree is better.

Once the subtree to rebuild is chosen and the resource allocation is determined, the problem becomes a task for a contract algorithm. A good candidate for such an algorithm is LSID3, which is expected to produce better subtrees when invoked with a higher resource allocation. LSID3 receives its resource allocation in terms of r, the sample size. Given a tree node y, we can view the task of rebuilding the subtree below y as an independent task. Every time y is selected, we have to allocate resources for the repair process. Following Russell & Zilberstein (1996), the optimal strategy is to double the amount of resources at each iteration. Thus, if the resources allocated for the last attempted improvement of y were LAST-R(y), the next allocation will be NEXT-R$(y) = 2 \cdot$ LAST-R(y). Note that when a node y is chosen for the first time LAST-R$(y) = 0$. In this case the next allocation will be NEXT-R$(y) = 1$.

Intuitively, the next node we would like to improve is the one with the highest expected marginal utility (Hovitz 1990; Russell & Wefald 1989), i.e., the one with the highest ratio between the expected benefit and the expected cost. Predicting the expected benefit and expected cost of rebuilding a subtree is a difficult problem. In what follows we show how to approximate these values, and how to incorporate these approximations into the node selection procedure.

Expected Cost: The expected cost can be approximated using the average time complexity of the contract algorithm used to rebuild subtrees. Following Equation 1, we estimate the expected runtime of LSID3(r) for rebuilding the subtree below a node y by NEXT-R$(y) \cdot m \cdot n^3$, where m is the number of examples that reach y and n is the number of attributes to consider. We observe that subtrees rooted in deeper levels are expected to be less costly because they have fewer examples and attributes to consider. Thus, their expected runtime is shorter. Further, because each time allocation for a node doubles the previous one, nodes that have already been selected many times for improvement will have higher associated costs and are less likely to be chosen again.

Granularity: Considering the cost approximation described above, the selection procedure may repeatedly attempt to improve subtrees rooted at deep nodes due to their low associated costs. This behavior would indeed be beneficial in the short term, but can be harmful in the long term. This is because when the algorithm later improves subtrees in upper levels, the resources spent on deeper nodes are wasted. Had the algorithm first selected the upper level trees, this waste would have been avoided, but the time gaps between potential improvements would have increased. To control the tradeoff between efficient resource use and anytime performance flexibility, we add a *granularity parameter* $0 \leq g \leq 1$. This parameter serves as a threshold for the time allocation for an improvement phase. A node can be selected for improvement only if its normalized expected cost is above g. To compute the normalized expected cost, we divide the expected cost by the expected cost of the root.

Expected benefit: Following Occam's Razor, the size of a subtree can serve as a measure for its quality. It is difficult, however, to know the size of the reconstructed subtree without actually building it. Therefore, we use instead an upper limit on the possible reduction in size. The minimal size possible for a decision tree is obtained when all examples are labelled with the same class. Such cases are easily recognized by the greedy ID3. Similarly, if a subtree were replaceable by another subtree of depth 1, ID3 (and LSID3) would have chosen the smaller subtree. Thus, the maximal size reduction of an existing subtree is to a depth of 2. Assuming that the maximal number of values per attribute is b, the maximal size of such a tree (measured by the number of leaves) is b^2. Hence, an upper bound on the benefit from reconstructing an existing tree t is SIZE$(t) - b^2$. If the expected costs are ignored and only the expected benefits considered, the highest score would always be given to the root node. This makes sense: assuming that we have infinite resources, we would attempt to improve the entire tree rather than parts of it. Note that there may be nodes whose cost is higher than the cost of the root node, since the expected cost doubles the cost of the last improvement of the node. Therefore, the normalized expected cost can be higher than 1. Such nodes,

Procedure TS-GREEDY-BUILD-TREE(E, A)
  Return ID3(E, A)

Procedure TS-REBUILD-TREE(E, A, r)
  Return LSID3(E, A, r)

Procedure TS-EXPECTED-COST$(T, node)$
  Return NEXT-R$(node) \cdot |$EX$(node)| \cdot |$ATT$(node)|^3$

Procedure TS-EXPECTED-BENEFIT(T)
  l-bound $\leftarrow (\min_{a \in A_{node}} |$DOMAIN$(a)|)^2$
  Return TREE-SIZE(T) $-$ *l-bound*

Procedure TS-BETTER(T_1, T_2)
  Return TREE-SIZE$(T_1) <$ TREE-SIZE(T_2)

Figure 5: IIDT-TS

however, will never be selected for improvement, because their expected benefit is necessarily lower than the expected benefit of the root node. Hence, when $g = 1$, IIDT is forced to choose the root node and its behavior becomes identical to that of the sequencing method.

Evaluating a subtree: Although LSID3 is expected to produce better trees when allocated more resources, an improved result is not guaranteed. Thus, to avoid obtaining a tree of lower quality, IIDT replaces an existing subtree with a newly induced alternative only if the latter is expected to improve the quality of the complete tree. As in the expected benefit approximation, we measure the usefulness of a tree by its size. Only if the reconstructed subtree is smaller does it replace an existing subtree. This guarantees that the size of the complete decision tree will decrease monotonically.

Figure 4 formalizes the node selection component in IIDT. We refer to the above-described instantiation of IIDT that uses the tree size as a quality metric by *IIDT-TS*. Figure 5 formalizes IIDT-TS.

Pruning the IIDT Trees

Pruning tackles the problem of how to avoid overfitting the data, mainly in the presence of classification noise. Pruning, however, is not able to overcome wrong decisions made when learning hard concepts. Therefore, our anytime approach and pruning address different problems: whereas pruning attempts to simplify the induced trees to avoid overfitting, IIDT attempts to exploit additional time for better learning of hard concepts. We view pruning as orthogonal to lookahead and suggest considering it to avoid overfitting.

Pruning can be applied to the trees produced by IIDT in two different ways. The first is to post-prune a tree when it is returned to the user. In this case IIDT maintains consistent trees and only when interrupted, it applies a post-pruning phase. When resumed, IIDT continues to improve the unpruned tree. This solution requires the user take to into account the delay imposed by pruning. In the anytime setup, where much time is invested for growing the tree, this delay is usually negligible with respect to the growing time.

Procedure EE-GREEDY-BUILD-TREE(E, A)
  Return C4.5(E, A)

Procedure EE-REBUILD-TREE(E, A, r)
  Return LSID3-P(E, A, r)

Procedure EE-EXPECTED-COST$(T, node)$
  Return NEXT-R$(node) \cdot |$EX$(node)| \cdot |$ATT$(node)|^3$

Procedure EE-EXPECTED-BENEFIT(T)
  l-bound $\leftarrow 0$
  Foreach $c \in C$
    $E_c \leftarrow \{e | e \in E_T,$ CLASS$(e) = c\}$
    l-bound $\leftarrow$ *l-bound* $+$ EXPECTED-ERROR(E_c)
  Return EXPECTED-ERROR(E_T) $-$ *l-bound*

Procedure EE-BETTER(T_1, T_2)
  Return $\sum_{l \in T_1}$ EXPECTED-ERROR$(E_l) <$
    $\sum_{l \in T_2}$ EXPECTED-ERROR(E_l)

Figure 6: IIDT-EE

The second alternative we propose maintains (possibly) inconsistent trees by building a pruned initial tree (using C4.5 (Quinlan 1993)), and by post-pruning each of the replacement trees (using LSID3-p). In this case the tree can be returned immediately, without any post-pruning delay.

Using the size of a tree as a measure for its quality is not applicable to inconsistent trees: one can easily create a tree of size 1 when consistency is not forced. One possible solution is to evaluate the quality of a subtree by its *expected error*. The expected error of a tree is measured by summing up the expected errors of its leaves, exactly as in C4.5's error-based pruning. We denote this instantiation of IIDT by *IIDT-EE*. In IIDT-EE, the expected benefit of rebuilding a subtree is measured by the maximal possible reduction in the expected error. A trivial lower bound for the expected error is the error if all examples were immediately split into subsets according to their label. Figure 6 lists IIDT-EE.

Incremental IIDT

As an anytime algorithm, IIDT continuously invests resources in attempt to improve the induced tree. It is not designed, however, to exploit additional data that becomes available with time. One way to handle examples that are presented incrementally is to rerun the inducer when a new example arrives. The restart approach is very reasonable for greedy learners, where the time "wasted" by starting the induction process from scratch is minimal. This straightforward conversion, however, is much more problematic in the case of anytime learners: it is not reasonable to discard the current hypothesis, which was produced using significant resources, just because a new example arrives. Greedy restructurings, as suggested in ID5R (Utgoff 1989) and its successive ITI (Utgoff, Berkman, & Clouse 1997), allow to quickly obtain the same greedy tree as in the batch setup. Applying such corrections to IIDT, however, will nullify its

```
...
E_new ← GET-NEW-EXAMPLES()
Foreach e ∈ E_new
   ADD-EXAMPLE(T, E, A, e)
E ← E ∪ {e}
Return T
```

Figure 7: I-IIDT (only the lines added to IIDT are listed)

```
Procedure TS-ADD-EXAMPLE(T, E, A, e)
  l ← LEAF-OF(e)
  T_l ← ID3(T, EX(l), ATT(l))
  replace l with T_l
  Δsize ← TREE-SIZE(T_l) − 1
  Foreach p ancestor of T_l
    TREE-SIZE(p) ← TREE-SIZE(p) + Δsize
```

Figure 8: Incorporating a new example by I-IIDT-TS

efforts to make better split decisions. To overcome these problems, we propose adapting IIDT to handle incremental learning tasks.

We assume that a stream of examples is being received and stored and that the learner can process the accumulated examples at any time. The most natural point to do so is upon the termination of each improvement iteration. In Figure 7 we generalize IIDT for incremental tasks. The incremental version of IIDT, called *I-IIDT*, is identical to the non-incremental IIDT listed in Figure 2 with one exception: at the end of each iteration, the algorithm takes the new examples and use them to update the tree. We propose two methods for tree update—one for each instantiation of IIDT.

Our first method for tree update is intended for IIDT-TS. Recall that IIDT retains for each node y the set of examples belongs to it, E_y. Given a new example e, and the current tree T, we search for the leaf l that e belongs to. If the label of l differs from that of e, we replace l by a consistent tree, built from $E_l \cup \{e\}$. This tree can be quickly obtained by invoking ID3. Such an update, that replaces a leaf l with a subtree, increases the size of the subtrees under all the nodes along the path from the root to l. Consequently, the expected benefit from rebuilding these subtrees increases. On the other hand, the expected cost, which depends on the number of examples, will increase as well. The selection mechanism of I-IIDT which prefers subtrees with high expected utility will take both changes into consideration. We refer to this variant as *I-IIDT-TS*. Figure 8 lists I-IIDT-TS.

The second method is designed for IIDT-EE. With the possibility to maintain inconsistent trees, adding a new example to the existing tree becomes as simple as finding the relevant leaf and updating the expected error of this leaf and of its ancestors. Clearly, a new example that is consistent with the current tree reduces the expected error, while an inconsistent example increases it. This effect is the strongest in the leaf and diminishes for shallower subtrees. We refer to this variant as *I-IIDT-EE*. Figure 9 formalizes I-IIDT-EE.

```
Procedure EE-ADD-EXAMPLE(T, E, A, e)
  l ← LEAF-OF(e)
  error ←EXPECTED-ERROR(EX(l))
  E_l ← EX(l) ∪ {e}
  error′ ←EXPECTED-ERROR(E_l)
  Δerror ← error − error′
  Foreach p ancestor of l
    EXPECTED-ERROR(EX(p)) ←
        EXPECTED-ERROR(EX(p)) + Δerror
```

Figure 9: Incorporating a new example by I-IIDT-EE

Experimental Evaluation

A variety of experiments were conducted to test the performance and anytime behavior of IIDT. First we describe a batch-mode comparison of IIDT and the fixed-time algorithms ID3 and C4.5.[3] Then we compare the batch IIDT to several modern decision tree induction methods. Finally we examine the performance of IIDT in the incremental setup.

Anytime Behavior of IIDT

In our first set of experiments, two versions of IIDT were evaluated: IIDT-TS(1), parameterized with a granularity factor 1 and thus behaves exactly as the sequencing method, and IIDT-TS(0.1), where the granularity factor is set to 0.1. Both versions do not prune the final tree.

The behavior of anytime learners on easy concepts is less interesting because the greedy algorithms are able to produce good trees with few resources. Therefore, we present here the results for more complex concepts that can benefit from larger resource allocation: the *Glass* and the *Tic-Tac-Toe* UCI datasets (Blake & Merz 1998), the *20-Multiplexer* dataset (Quinlan 1993), and the 10-XOR dataset, generated with additional 10 irrelevant attributes.[4] The performance of the different algorithms is compared both in terms of generalization accuracy and size of the induced trees, measured by the number of leaves. Following the recommendations of (Bouckaert 2003), 10 runs of 10-fold cross-validation experiments were conducted for each dataset.

Figures 10 shows the anytime graphs for both tree size and accuracy for the 4 datasets. Each graph represents an average of 100 runs (for the 10×10 cross validation). In all cases both anytime versions indeed exploit the additional resources and produce smaller and more accurate trees.[5] Because our algorithm replaces a subtree only if the new one is smaller, all size graphs decrease monotonically. The most interesting anytime behavior is for the difficult *10-XOR* problem. There, the tree size decreases from 4000 leaves to almost the optimal size 2^{10}, and the accuracy increases from 50% (which is the accuracy achieved by C4.5) to al-

[3] C4.5 was run with its default parameters (with information gain replacing the default gain ratio).

[4] The artificial datasets are publicly available at: http://www.cs.technion.ac.il/∼esaher/publications/datasets.

[5] Note that in some cases C4.5 produces smaller, yet less accurate trees because it allows inconsistency by pruning the tree.

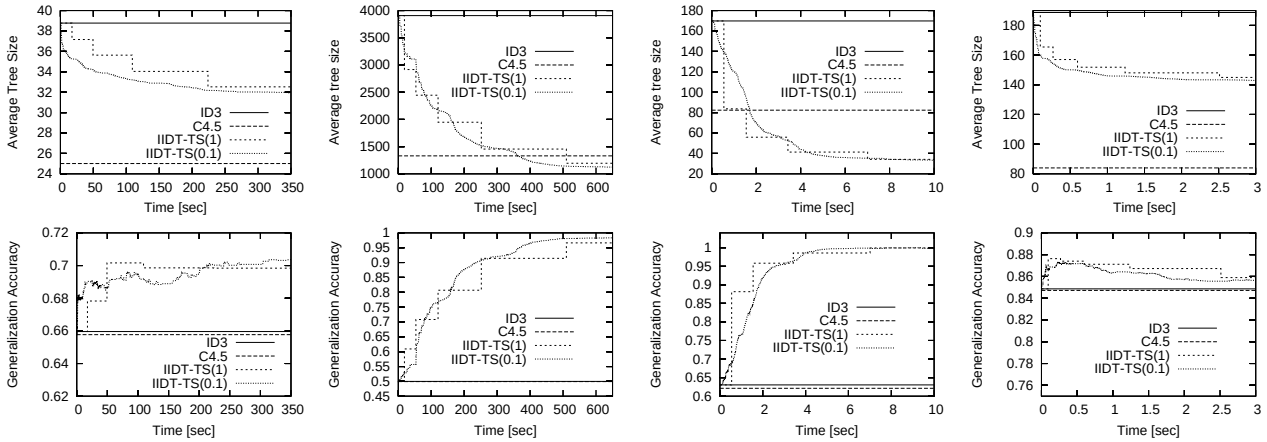

Figure 10: Anytime behavior on the Glass (leftmost), XOR10, Multiplexer20, and Tic-tac-toe (rightmost) datasets

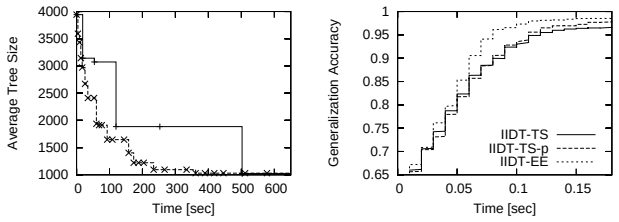

Figure 11: (left) Time steps for an arbitrary run on *10-XOR* (right) Pruned IIDT trees on the noisy *XOR-5* problem.

most 100%. The shape of the graphs is typical of anytime algorithms with diminishing returns.

The improvement in the accuracy of IIDT-TS (at the latest point it was measured) over ID3 and C4.5 was found by t-test ($\alpha = 0.05$) to be significant for all datasets except of the *Tic-tac-toe*. The performance of IIDT-TS on *Tic-tac-toe* slightly degrades over time. We believe that this because it is difficult to represent the *Tic-tac-toe* concept with the given primitive features by a single tree. Indeed, Markovitch & Rosenstein (2002) showed that when using slightly richer features, ID3 could easily solve the problem.

The difference in performance for the two anytime algorithms is interesting. IIDT-TS(0.1), with the lower granularity parameter, indeed produces smoother anytime graphs (with lower volatility) which allows better control and better predictability of return. Moreover, in large portions of the time axis, the IIDT-TS(0.1) graph dominates that of IIDT-TS(1) due to its more sophisticated node selection. The smoothness of the IIDT-TS(0.1) graph is somehow misleading since it represents an average of 100 runs with steps at different time points. This is in contrast to the IIDT-TS(1) graph, where the steps are roughly at the same time points. Figure 11 (left) shows one anytime graph (out of the 100). We can see that although the IIDT-TS(0.1) graph is less smooth than the average, it is still much smoother than the corresponding IIDT-TS(1) graph.

To test the performance of the the pruned IIDT variants we used the *XOR-5* problem with artificially added noise. Figure 11 (right) shows the results for non-pruned IIDT-TS, pruned IIDT-TS (denoted by IIDT-TS-p) and IIDT-EE. In all cases g was set to 0.1. The best performance is that of IIDT-EE.

Comparison with Modern Decision Tree Learners

Several decision tree learners have been introduced during the last decade. Although these learners were not presented and studied as anytime algorithms, some of them can be viewed as such. In what follows we compare the performance of IIDT-TS(0.1) on the hard *XOR-5* concept (with additional 5 irrelevant attributes) to that of Skewing, GATree, Bagging and ADTree. We first give a brief overview of the studied methods and then compare their performance to that of our anytime approach.

Page & Ray (2004) introduced *Skewing* as an alternative to lookahead for addressing problematic concepts such as parity functions. At each node, the algorithm skews the set of examples and produces several versions of it, each with different weights for the instances. The attribute that exceeds a pre-set gain threshold for the greatest number of weightings is chosen to split on. Skewing was reported to successfully learn hard concepts such as XOR-n mainly when the dataset is large enough relative to the number of attributes. Skewing can be viewed as a contract algorithm parameterized by w, the number of weightings. To make it interruptible, we apply the sequencing approach and run it with exponentially increasing values of w. In our experiments, we also consider *Sequential Skewing*, which skews one variable at a time instead of all of them simultaneously. This version is not parameterized. To convert it into an anytime algorithm, we added a parameter k that controls the number of Skewing iterations. Thus, instead of skewing each variable once, we skew it k times.

Papagelis & Kalles (2001) presented *GATree*, a learner that uses genetic algorithms to evolve decision trees. The initial population consists of randomly generated 1-level

depth decision trees, where the test and the class labels are drawn randomly. The mutation step chooses a random node of a desired tree and replaces the node's test (or label) with a new random one. The crossover step chooses two random nodes, and swaps their subtrees. When tested on several UCI datasets, GATree was reported to produce trees as accurate as C4.5 but of significantly smaller size. GATree can be viewed as an interruptible anytime algorithm that uses additional time to produce more and more generations. In our experiments we use GATree with the same set of parameters as reported by Papagelis & Kalles (2001), with one exception: we allowed a larger number of generations.

The third algorithm we tested is *Bagging* (Breiman 1996). Bagging forms a committee of trees by making bootstarp replicates of the training set and using each one to learn a decision tree. Bagging is an ensemble-based method, and as such, it is naturally an interruptible anytime learner. Additional resources can be exploited to generate larger committees. In our experiments we consider Bagging methods that use ID3 and C4.5 as base learners. Note that Bagging is not a direct competitor to our method. We defined our goal as inducing a single "good" decision tree while bagging generates a set of trees. Generating a set of trees rather than a single good tree eliminates one of the greatest advantages of decision trees—their comprehensibility.

ADTree (Freund & Mason 1999) attempts to address the incomprehensibility of ensembles by combining Boosting with a new representation for decision trees, called Alternating Trees. In our experiments we used the implementation of ADTree in WEKA (Witten & Frank 2005).

Figures 12 plots the results for IIDT-TS, Skewing, and Bagging. Since GATree and ADTree were run on different environments, we report their results separately later on. The graphs show that IIDT-TS clearly outperforms the other methods both in terms of tree size and accuracy. It reaches almost perfect accuracy (99%), while Bagging-ID3 and Skewing top at 55%.

The inferior performance of Bagging-ID3 is not surprising. The trees that form the committee were induced greedily and hence could not discover these difficult concepts, even when they were combined. Similar results were obtained when running Bagging over C4.5.

The inferior results of the Skewing algorithms are more difficult to interpret, since Skewing was shown to handle difficult concepts well. One possible explanation for this is the small number of examples with respect to the difficulty of the problem. To verify that, we repeated the experiment with simpler *XOR* problems such as *XOR-2* and *XOR-3*. In these cases Skewing indeed did much better and outperformed ID3, reaching 100% accuracy (similarly to IIDT-TS). When we increased the size of the training set for the *XOR-5* domain, Skewing also performed better, yet IIDT-TS outperformed it by more than 9%.

The average accuracy of GATree, after 150 generations (almost 3 seconds), was 49.5%. Thus, even when allocated much more time than IIDT-TS, GATree could not compete with it. We repeated the experiment, allowing GATree to have a larger initial population. The accuracy on the testing set, even after thousands of generations, remained very low.

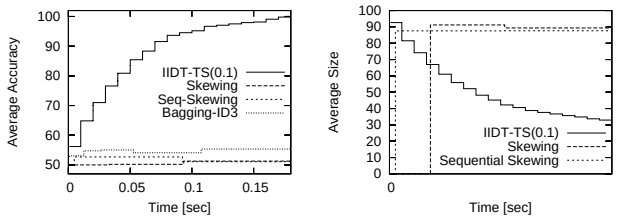

Figure 12: Performance of various algorithms on *XOR-5*

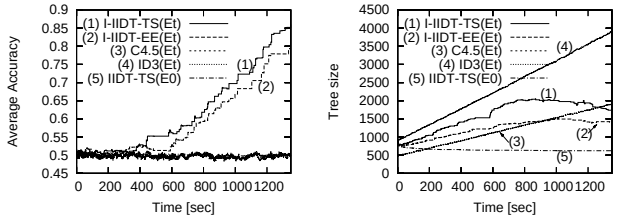

Figure 13: Learning the XOR-10 problem incrementally

ADTree, also could not learn the concept and produced classifiers of low accuracy (59%), even after 1000 Boosting iterations (more than 4 seconds).

The above set of experiments was repeated on the much more difficult *XOR-10* dataset. The advantage of IIDT-TS was even more evident. While IIDT-TS was able to reach accuracy of 99%, Bagging, Skewing, GATree, and ADTree performed poorly as a random guesser, with 50% accuracy.

IIDT as Incremental Learner

The two instantiations of I-IIDT, namely I-IIDT-TS(0.1) and I-IIDT-EE(0.1) are compared to ID3^R, C4.5^R, and the non-incremental IIDT. The suffix "R" means that the algorithm is restarted with the arrival of a new example. In the case of the fixed time algorithms, ID3 and C4.5, we ignore the runtime and assume that each tree is built within 0 seconds. Since the majority of incremental methods proposed in the literature attempt to produce the same tree as ID3 or C4.5 would have produced given all the examples, ID3^R and C4.5^R give an upper bound on the performance of these incremental methods. To simulate an incremental setup, we randomly partition each training set into 25% initial set and 75% incremental set. Note that unlike the other methods, the non-incremental IIDT gets only the initial set of examples.

Figures 13 shows the results for the *XOR-10* datsets. Both versions of I-IIDT exhibit good anytime behavior. They exploit the additional time and examples. Their advantage over the greedy algorithms ID3 and C4.5, and over the batch anytime algorithm is clear. These results demonstrate the need for combining anytime and incremental learning: the anytime incremental learner succeeds where the greedy incremental inducers and the anytime batch learner fail. Comparing the performance of I-IIDT-TS to that of I-IIDT-EE shows that IIDT-TS is better for this non-noisy setup. When we repeated the same experiment with artificially added noise I-IIDT-EE was superior, due to its ability to avoid overfitting.

Conclusions and Related Work

In this work we explored the problem of how to produce better decision trees when more time resources are available. Unlike the contract setup that was addressed in a previous study, this work does not assume a priori knowledge of the amount of the resources available and allows the user to interrupt the learning phase at any moment.

The major contribution of this paper is the IIDT framework, which can be adjusted to use any contract algorithm for reproducing a decision tree and any measure for choosing the subtree to rebuild. Moreover, IIDT can be configured to receive new training examples and is therefore appropriate for incremental learning tasks. The experimental study shows that IIDT exhibits good anytime behavior, allowing computation speed to be traded for a higher-quality induced hypothesis.

While, to the best of our knowledge, there has been no specific attempt to design an anytime interruptible algorithm for decision tree induction, there are several related works, aside from those mentioned in the previous section, that warrant discussion here. Utgoff, Berkman, & Clouse (1997) presented DMTI, an induction algorithm that uses a direct measure of tree quality instead of a greedy heuristic to evaluate the possible splits. Several possible tree measures were examined and the MDL measure was shown to have the best performance. DMTI can use a fixed amount of additional resources and hence cannot serve as an interruptible anytime algorithm. Furthermore, DMTI uses the greedy approach to produce the lookahead trees which may be insufficient to accurately estimate the usefulness of a split. Opitz (1995) introduced an anytime approach for theory refinement. This approach starts by generating a neural network from a set of rules, and then attempts to improve the resulting hypothesis using the training data. Lizotte, Madani, & Greiner (2003) defined the related problem of *budgeted learning*, where a cost is associated with obtaining each attribute value of a training example, and the task is to determine what attributes to test given a budget.

In the future we intend to improve the performance of IIDT by using information from past improvement attempts. In addition, we plan to apply monitoring techniques for optimal scheduling of IIDT and to examine other strategies for choosing nodes and for improving subtrees.

Acknowledgements

This work was partially supported by funding from the EC-sponsored MUSCLE Network of Excellence (FP6-507752).

References

Blake, C. L., and Merz, C. J. 1998. UCI repository of machine learning databases.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's Razor. *Information Processing Letters* 24(6):377–380.

Bouckaert, R. R. 2003. Choosing between two learning algorithms based on calibrated tests. In *ICML'03*, 51–58.

Breiman, L. 1996. Bagging predictors. *Machine Learning* 24(2):123–140.

Craven, M. W. 1996. *Extracting Comprehensible Models from Trained Neural Networks*. Ph.D. Dissertation, University of Wisconsin, Madison.

Domingos, P. 1999. The role of Occam's Razor in knowledge discovery. *Data Mining and Knowledge Discovery* 3(4):409–425.

Esmeir, S., and Markovitch, S. 2004. Lookahead-based algorithms for anytime induction of decision trees. In *ICML'04*, 257–264.

Freund, Y., and Mason, L. 1999. The alternating decision tree learning algorithm,. In *ICML'99*, 124–133. Morgan Kaufmann, San Francisco, CA.

Hastie, T.; Tibshirani, R.; and Friedman, J. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag.

Hovitz, E. 1990. *Computation and Action under Bounded Resources*. Ph.D. Dissertation, Computer Science Department, Stanford University.

Hyafil, L., and Rivest, R. L. 1976. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters* 5(1):15–17.

Lizotte, D. J.; Madani, O.; and Greiner, R. 2003. Budgeted learning of naive Bayes classifiers. In *UAI'03*.

Markovitch, S., and Rosenstein, D. 2002. Feature generation using general constructor functions. *Machine Learning* 49:59–98.

Murphy, O. J., and McCraw, R. L. 1991. Designing storage efficient decision trees. *IEEE-TC* 40(3):315–320.

Opitz, D. W. 1995. *An Anytime Approach to Connectionist Theory Refinement: Refining the Topologies of Knowledge-Based Neural Networks.* Ph.D. Dissertation, Department of Computer Sciences, University of Wisconsin-Madison.

Page, D., and Ray, S. 2004. Sequential skewing: an improved skewing algorithm. In *ICML'04*, 86–93.

Papagelis, A., and Kalles, D. 2001. Breeding decision trees using evolutionary techniques. In *ICML'01*, 393–400.

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

Russell, S. J., and Wefald, E. 1989. Principles of metareasoning. In *KR'89*, 400–411.

Russell, S., and Zilberstein, S. 1996. Optimal composition of real-time systems. *Artificial Intelligence* 82(1):181–213.

Shavlik, J. W.; Mooney, R. J.; and Towell, G. G. 1991. Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning* 6(2):111–143.

Utgoff, P. E.; Berkman, N. C.; and Clouse, J. A. 1997. Decision tree induction based on efficient tree restructuring. *Machine Learning* 29(1):5–44.

Utgoff, P. E. 1989. Incremental induction of decision trees. *Machine Learning* 4(2):161–186.

Vapnik, M. A. 1995. *The Nature of Statistical Learning Theory*. New York: Springer-Verlag.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, CA.