A Study on Efficient Generation of Decision Trees Using Genetic Programming

Toru Tanigawa
Department of Computer Software
The University of Aizu
Aizu-Wakamatsu, 965-8580, Japan
s1041123@u-aizu.ac.jp
Tel:+81-242-24-3696

Qiangfu Zhao
The University of Aizu
Aizu-Wakamatsu, 965-8580, Japan
qf-zhao@u-aizu.ac.jp
Tel:+81-242-37-2519
Fax:+81-242-37-2743

Abstract

For pattern recognition, the decision trees (DTs) are more efficient than neural networks (NNs) for two reasons. First, the computations in making decisions are simpler. Second, important features can be selected automatically during the design process. On the other hand, NNs are adaptable, and thus have the ability to learn in changing environment. Noting that there is a simple mapping from DT to NN, we can design a DT first, and then map it to an NN. By so doing, we can integrate the symbolic (DT) and the sub-symbolic (NN) approaches, and have advantages of both. For this purpose, we should design DTs which are as small as possible. In this paper, we continue our study on the evolutionary design of the decision trees based on genetic programming, and propose two new methods to reduce the tree sizes. The effectiveness of the new methods are tested through experiments with a character recognition problem.

1 Introduction

It is known that decision trees (DTs) are very efficient for pattern recognition (Meisel, 1973), (Gelfand, 1991) and (Henrichon, 1969)). Actually, as long as pattern recognition is considered, a DT is more efficient than a neural network (NN). There are mainly two reasons. First, the computations in making decisions are simpler — only one feature is used in each non-terminal (hidden) node, and the only computation can be a very simple comparison (say, $x_i < a$?). Second, important features can be selected automatically during the design process. Actually, when many features are provided, only a few of them are often useful for making correct decisions. In using an NN, since we do not know which feature is important, the only thing we can do is to use all features.

However, the DTs do not have the adaptive or learning ability, and thus they cannot be used in changing environment. This problem can be avoided if we map a DT to an NN. Actually, there is a very simple mapping from DT to NN (Sethi, 1990). This mapping integrates the symbolic approach (DTs) and the sub-symbolic one (NNs). Specifically, it makes DTs adaptable, and at the same time, provides a systematic way for structural learning of NNs. In addition, since the features are well selected, the NNs obtained from this mapping may have much fewer connections than those designed directly. The key point here is to design a DT which is as small as possible. To design an optimal DT, however, is not so easy. It has been proved that designing optimal binary decision trees is an NP-complete problem (Hyafil, 1976).

Recently, many authors have tried to solve different kinds of NP-complete problems using the evolutionary approach. In our study, we have also used genetic programming (GP) to the design of DTs (Shirasaka, 1998) and (Zhao, 1999). Since all kind of DTs can be reduced to binary decision trees (BDTs), we consider BDTs only. The GP used in our research is the same as the original algorithm given in (Koza, 1994) except that the goal is different. Therefore, most of the discussions given here may be useful also for the study of general GP.

The purpose of this paper is to study how to use GP to produce BDTs with smaller sizes. Various methods have been proposed in the literature to reduce the sizes of the BDTs or the computer programs produced by GP (Iba, 1994) and (Zhao, 1999), but results obtained up to now are still not good enough to apply GP to neural network learning. In this paper, we propose

two new approaches based on the concept of *divide-and-conquer*. The first one is a *bottom-up* approach, which tries to divide the pattern space into many small parts first, and then merge them into larger and larger spaces. The second approach is *top-down*, which tries to divide the pattern space into two smaller parts, and then four, and so on. In both approaches, the idea is to design small BDTs first, and then use them as sub-programs in designing larger and more powerful BDTs. The effectiveness of the new methods will be tested through experiments with a simple handwritten character recognition problem.

2 A Brief Review of the Binary Decision Tree

In our study, a binary decision tree (BDT) is defined as a list of 7-tuples. Each 7-tuple corresponds to a node. There are two kinds of nodes: *non-terminal node* and *terminal node*. Specifically, a node is defined by

$$node = \{t, label, P, L, R, C, size\}$$

where

- t is the node number. The node with number $t = 0$ is called the *root*.
- *label* is the class label of a terminal node, and it is meaningful only for terminal nodes.
- P is a pointer to the parent. for the root, P=NULL.
- L and R are the pointers to the left and the right children, respectively. For a terminal node, both pointers are NULL.
- C is a set of registers. For a non-terminal node, $n = C[0]$ and $a = C[1]$, and the classification is made using the following comparison:

$$feature_n < a? \tag{1}$$

  If the result is YES, visit the left child; otherwise, visit the right child. For a terminal node, $C[i]$ is the number of training samples of the i-th class, which are classified to this node. The label of a terminal node is determined by majority voting. That is, if

$$C[k] = \max_{\forall i} C[i] \tag{2}$$

  then, $label = k$.

- *size* is the size of the node when it is considered as a *sub-tree*. This parameter is useful for selecting good trees during evolution. The size of the root is the size of the whole tree, and the size of a terminal node is 1.

Many results have been obtained during the last two decades for construction of BDTs (Meisel, 1973), (Gelfand, 1991),(Quinlan, 1986) and (Henrichon, 1969). To construct a BDT, it is assumed that a *training set* consisting of feature vectors and their corresponding class labels are available. The BDT is then constructed by partitioning the feature space in such a way as to recursively generate the tree. This procedure involves three steps: splitting nodes, determining which nodes are terminal nodes, and assigning class labels to terminal nodes. Among them, the most important and most time consuming step is splitting the nodes. Actually, if we want to find the best feature for splitting each node, and find the optimal BDT for a given training set, the problem is NP-complete (Hyafil, 1976).

As pointed out earlier, if we adopt the evolutionary algorithms (EAs) in designing the BDTs, we may find much better solutions than conventional methods. In the recent years, many authors have used EAs to solve different kinds of NP-complete problems. Among many EAs, GP is most suitable for designing BDTs, because the genotype can be considered as a decision tree directly.

3 Evolutionary Design of Binary Decision Trees

The algorithm used for designing BDTs is almost the same as GP, except that the goal is different. Therefore, GP can be adopted with minor revision. On the other hand, many of the discussions given here may be useful also for the study of general GP.

The over-all evolution process is the same as the standard GA. Similar to GP, an individual is defined as a BDT, which represents both the genotype and the phenotype. These individuals can be initialized using a $size > 1$. In the experiments given in this paper, we often initialize the trees with size=3, 5, or 7.

The fitness of a decision tree is defined by

$$fitness = \frac{N_{correct}}{N_{total}} \tag{3}$$

where $N_{correct}$ is the number of correctly classified samples, and N_{total} is the total number of training samples. The BDTs designed based on this definition, however, are usually very large. These kind of trees are not optimal in any sense, and cannot be used as the

prototypes for further design of NNs. In this paper, we will investigate some methods for reducing the tree size. These methods will be discussed in the following sections.

The above fitness can be calculated as follows. First, present all training samples to an individual and classify them. If a sample x is classified to a terminal *node*, then

$$\text{node.C[label(x)]++} \tag{4}$$

where a C-language like expression is used. Second, determine the label of each terminal node by majority voting. Finally, find the fitness of a BDT by

$$fitness = \frac{1}{N_{total}} \sum_{\forall Terminals} \text{node.C[node.label]}. \tag{5}$$

During evolution, evaluation, selection, crossover and mutation are performed iteratively, until some stopping criterion is satisfied. For the selection operation, we can adopt any existing method. In our study, we just adopt the following simple strategy (truncate selection): in every generation, $r_s \times N$ of the individuals with the lowest fitness are replaced by offspring of the better individuals, where r_s is the selection rate, and N is the population size.

The crossover operation is the same as that used in the standard GP. That is, for two parents (selected at random), choose one sub-tree (including the whole tree) at random for each parent, and then exchange the sub-trees. The mutation operator is the same as that used in the GA. Note that in designing a BDT, it is not necessary to perform mutation on the terminal nodes. For each nonterminal node, $C[0]$ and $C[1]$ are encoded as binary strings. The mutation operator just reverses each bit according to a given rate. Because the number of features and the range of the feature values are usually different, $C[0]$ and $C[1]$ should have different word (gene) length.

4 Methods for Reducing Tree Sizes

Reducing the solution size is also an important topic in the study of general GP. For example, we can put the *description length* as a penalty in the fitness function, and try to find solutions with the minimum description length (MDL) (Iba, 1994). In our study, however, we do not adopt this method, because it is difficult to determine the weight of the penalty function.

In our study, we first proposed a simple method for reducing the solution size (Shirasaka, 1998). In this method, a size is assigned to each node (sub-tree) of a tree, and then the individuals are sorted according to the following rules:

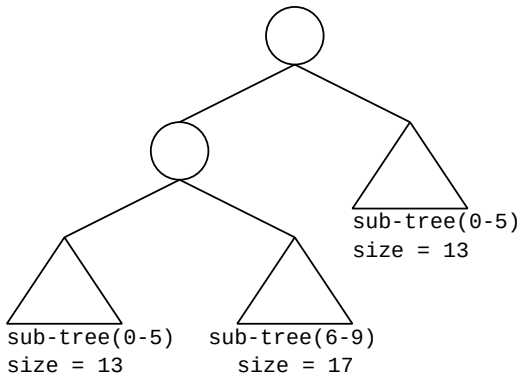

Figure 1: Divide a large tree into many

- if tree[i].fitness > tree[j].fitness, tree[i] is better,
- if tree[i].fitness = tree[j].fitness, **and** tree[i].size < tree[j].size, tree[i] is better.

Therefore, smaller trees will have higher probability to be selected. The results obtained by using the above method are much better than those obtained by using GP directly. However, the tree sizes are still not small enough for NN design. To reduce the tree sizes further, we also tried some other methods in (Zhao, 1999), but the results are not good enough either.

In this paper, we introduce two new approaches based on the concept of *divide-and-conquer*. The basic idea is to produce small sub-trees first, and then use them as sub-programs. The sub-programs can be re-used many times. This idea is based on the following observations:

- If we design a BDT using a large training set, the tree will be large.
- If we design a BDT to classify all patterns in the training set together, the tree will be large.

Thus, if we design a BDT for part of the training data, or for classifying some of the patterns, the tree will be smaller. Fig. 1 illustrates this idea. In this figure, a circle is a node, and a triangle is a sub-tree. The sub-trees are designed separately using GP for classifying some of the patterns in the training set. The sizes of the sub-trees are expected much smaller than that of the tree designed directly for classifying all patterns. In addition, the actual size of the tree in Fig. 1 is smaller than it looks. Although the total size is 2 +

$13 + 17 + 13 = 45$, the actual size is $2 + 13 + 17 = 32$ because a sub-tree can be re-used, and only one copy is necessary to realize this tree.

The above idea sounds interesting, but there is a key question here: how do we decompose the training set into many smaller sub-sets from which smaller BDTs can be designed ? To answer this question, we proposed two methods: the first one is a bottom-up approach, and the second is top-down. We will give detailed explanation for these two methods below.

4.1 Method-I: the bottom-up approach

To make the discussion more concrete, we use digit recognition as an example. In the bottom-up approach, we first design BDTs for classifying digits of only two different classes. For example, we can design BDTs for distinguishing all 0's from 1's, all 2's from 3's, and so on. These trees are denoted by BDT(0, 1), BDT(2, 3), and so on. Then, using them as sub-trees, we can design BDTs for classifying digits of four classes (See Fig. 2). For example, using BDT(0,1) and BDT(2,3) as sub-trees, we can design BDT(0, 1, 2, 3). This process can be summarized as follows:

1. Find a BDT for classifying 0 and 1 using GP, keep it as a sub-tree, which is denoted by BDT(0, 1). In the same way, find BDT(2, 3), BDT(4, 5), BDT(6, 7) and BDT(8, 9).
2. Find a BDT for classifying (0, 1) and (2, 3). Combine this tree with BDT(0, 1) and BDT(2, 3), we get another sub-tree, denoted by BDT(0, 1, 2, 3), which can classify 0, 1, 2, and 3. In the same way, we can get BDT(6, 7, 8, 9).
3. Find a BDT which can classify (0, 1, 2, 3) and (4, 5) using GP. Combine this tree with BDT(0, 1, 2, 3) and BDT(4, 5), we get BDT(0, 1, 2, 3, 4, 5).
4. Finally, find a BDT for classifying (0, 1, 2, 3, 4, 5) and (6, 7, 8, 9). Combine this tree with BDT(0, 1, 2, 3, 4, 5) and BDT(6, 7, 8, 9), we can get BDT(0, 1, 2, 3, 4, 5, 6, 7, 8, 9), which is the final result.

4.2 Method-II: the top-down approach

In this method, we divide the whole training set into two parts first. For example, we can find a BDT for classifying (0,1,2,3,4,5) and (6,7,8,9). Then, we find a BDT for classifying (0,1,2) and (3,4,5), and a BDT for (6,7) and (8,9). This process is continued until the current training set contains only patterns of the same class.

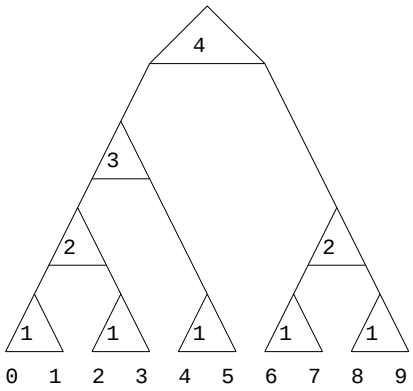

Figure 2: The bottom-up approach for designing a binary decision tree for digit recognition

In practice, however, we do not know how to divide the training set in advance. We just do this using GP. For any sub-tree, we provide all patterns in the *current training set*, and classify them into two categories: patterns who belong to the left-nodes, and those who belong to the right-nodes (see Fig. 3). Suppose that there are n patterns in the i-th class, n_1 of them are classified to the left-nodes, and $n_2 = n - n_1$ patterns are classified to the right-nodes. Then, all patterns of the i-th class will be classified to the left (right) category if $n_1 > n_2$ ($n_2 > n_1$). In this case, if a pattern of the i-th class is classified to a right (left) node, it is considered as a mis-classification.

To evaluate a sub-tree, we provide all training patterns (in the current training set) to the tree, assign each pattern to a proper category (left or right), and then count the number of correct classifications. The fitness is defined by

$$fitness = 1 - \frac{number\ of\ mis-classifications}{size\ of\ current\ training\ set}. \quad (6)$$

When we get a tree with fitness 1 (or approximately 1), we adopt GP to training samples classified to the left category and then to those belonging to the right category. This is done recursively until all samples in the current training set belong to the same class.

Note that each sub-tree (except the first one) can be used many times by its parent sub-tree if the size of the parent tree is larger than 3. The size of the whole decision tree is roughly the sum of sizes of all sub-trees obtained in the above recursive procedure.

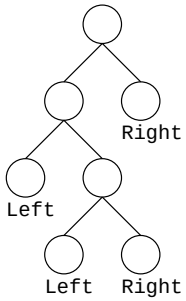

Figure 3: Divide all patterns into two categories

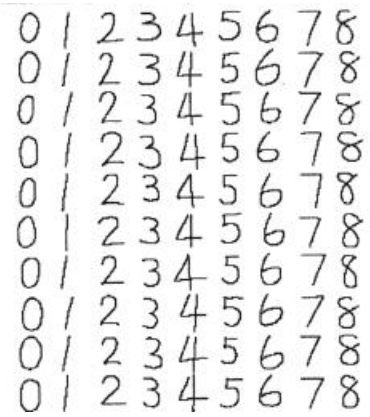

Figure 4: Part of training samples

5 Experimental Results

To verify the effectiveness of the methods proposed in this paper, we conducted some experiments with a simple digit recognition problem. In this problem, the training set contains 360 samples of 9 classes (40 samples per class). The digits were written using mouse in a 256 × 256 frame. Part of the samples are shown in Fig. 4. The digit 9 is not considered because features used in our experiment are rotational invariant (and thus 6 and 9 are the same). Specifically, the features are the crossing numbers on several concentric circles, with the center of the concentric circles being the center of gravity of the image.

5.1 Parameters

The parameters used in the experiments are given as follows:

- the number of features (the concentric circles) is 16.
- the gene length for $C[0]$ is 4 bit($2^4 = 16$).
- Since the value of the features are integers less than 8, the gene length for $C[1]$ is 3.
- The population size is 200.
- The selection rate is 0.4.
- The crossover rate is 1.0.
- The mutation rate is 0.01.
- Initially, the sizes of 60 of the individuals are 3, sizes of 70 of the individuals are 5, and sizes of all other individuals are 7.
- The number of generation is 5000.

In selection, $selection\_rate \times population\_size$ of the individuals are selected against, and they are replaced by offspring of other individuals. One point crossover is used.

5.2 Results of Method-I

Table 1 gives the results of 100 runs. In this table, $fitness$ is the recognition rate of the sub-trees for the specified sub-set of the training set, and $size$ is the size of the best sub-trees after 5,000 generations. The fitness and size are average values over 100 runs. $size_{min}$ is the minimum tree size obtained in 100 runs. Table 2 shows the results obtained by using the method (called **old-method** from now on) proposed in (Shirasaka, 1998). From these tables we can see that, in average, Method-I can get smaller decision trees. However, if we consider only the smallest tree in 100 runs, the old-method is better. Note that Table 1 actually shows the results obtained by combining Method-I and the old method.

5.3 Results of Method-II

Table 3 gives the experimental results of Method-II, where as in Table 1, $fitness$ and $size$ are average values over 100 runs, and $size_{min}$ is the size of the best tree obtained in 100 runs. Clearly, if we combine Method-II with the method proposed in (Shirasaka, 1998) (Method-II + Old in the table), we can always get very good results: more than 50 percent size reduction in average (compared with the old method). The minimum tree is also much smaller than that obtained using the old method.

6 Conclusion

From the experimental results we can see that both the bottom-up and the top-down approaches are, in average, very effective to reduce the tree size. Especially,

Table 1: The experimental results for Method-I

Sub-tree	$fitness$	$size$	$size_{min}$
BDT(0,1)	1	3	3
BDT(2,3)	0.999875	7	5
BDT(4,5)	1	3	3
BDT(6,7)	0.998125	9.6	7
BDT(0,1,2,3)	1	11.34	11
BDT(6,7,8)	1	3	3
BDT(0,1,2,3,4,5)	0.999667	32.92	21
BDT(0,1,2,3,4,5,6,7,8)	0.998056	19.6	13
Whole	0.995417	89.46	72

Table 2: The experimental results for old method

$fitness$	$size$	$size_{min}$
0.998555	117.32	55

Table 3: The experimental results for Method-II

Methods	$fitness$	$size_{ave}$	$size_{min}$
Method-II alone	0.993556	196.90	64
Method-II + Old	0.989278	50.46	34

by combining the top-down approach with the method proposed in (Shirasaka, 1998), we can always get very good results. Of course, it is still hard to make any general conclusion because the problem considered in this paper is still too simple. In the future, we would like to do more experimental study using larger database, and compare the GP approach with the conventional ones (say, C4.5 proposed by Quinlan). Maybe we can combine the conventional approaches with GP, and get much better solutions. In addition, we would like to map the results to neural networks, and discuss the adaptability and generalization ability before and after this mapping.

References

W. S. Meisel and D. A. Michalopoulos (1973). A partitioning algorithm with application in pattern classification and the optimization of decision tree. *IEEE Trans. on Computers* **22**(1): 93-103.

S. B. Gelfand, C. S. Ravishankar, and E. J. Delp (1991). An iterative growing and Pruning algorithm for classification tree design. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **13**(2): 163-174.

E. G. Henrichon, JR. and K. S. Fu (1969). A nonparametric partitioning procedure for pattern classification. *IEEE Trans. on Computers* **18**(7): 614-624.

I. K. Sethi (1990). Entropy nets: from decision trees to neural networks. *Proc. IEEE* **78**(10): 1605-1613.

L. Hyafil (1976). Construction optimal binary decision trees is NP-complete. *Information Processing Letters* **5**(1): 15-17.

J. R. Koza (1994). *Genetic Programming.* Fourth Printing. The MIT Press.

J. R. Quinlan (1986). Induction of decision trees. *Machine Learning* **1**: 81-106.

H. Iba, H. de Garis and T. Sato (1994). *Genetic programming using a minimum description length principle," Advances in Genetic Programming*, 265-283. MIT Press.

M. Shirasaka, Q. F. Zhao, O. Hammami, K. Kuroda and K. Saito (1998). Automatic design of binary decision trees based on genetic programming. *Proc. The Second Asia-Pacific Conference on Simulated Evolution and Learning.*

Q. F. Zhao and M. Shirasaka (1999). A Study on Evolutionary Design of Binary Decision Trees. *Proc. Congress on Evolutionary Computation*: 1988-1993.