STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVIII**, Number 2, 2003

EVOLVING ORTHOGONAL DECISION TREES

JOÓ ANDRÁS AND D. DUMITRESCU

Abstract. Instead of using or fine-tuning the well-known greedy methods to induce decision trees, we propose a new method, which explores the 'brute' force of evolutionary algorithms to evolve decision trees, used mainly for classification. MEP, a new evolutionary technique is used for representing the decision trees.

The paper is organized as follows: the introduction makes a short overview of the decision tree induction techniques. Section 2 contains the short review of the basics of the MEP. Section 3 contains the description of the MEPDTI. Section 4 presents the results, followed by the conclusions, and possible ways to improve the presented method.

Keywords: decision tree, evolutionary algorithms.

1. Introduction

Decision trees are one of the best-known classifiers systems. They classify *instances*, propagating the instance through the decision tree, testing at each internal node one or more *attributes* of the instance. When the instance reaches one of the final nodes of the tree, it is classified: the decision trees have in their leaf nodes the possible *classes* in which the instance can be classified. Decision trees are usually built using a set of *training instances*, called the *training data*, and tested with another set of instances, called the *test data*. The building process is referred as the decision tree *induction*.

Decision trees can be classified using several criteria. One criterion is the *number of classes*: if there are only two classes, they are called binary decision trees. Another criterion is the *type of attributes*, which characterize an instance. There are two main classes of attribute types: *symbolic* (unordered), and *numeric* (ordered). Another axis on which decision trees can be classified is the way in which they are built from the training data (the *tree induction type*). The classical way is a greedy method, on which the most of the tree induction algorithms are based. Another approach is the evolutionary method: the trees are not built, but evolved. But (probably) the main criteria by which the decision trees are categorized is the *test type*, which is made in the internal nodes of the decision tree. There are two

Received by the editors: June 2003.

2000 *Mathematics Subject Classification.* 62C99, 68T20.

1998 *CR Categories and Descriptors.* I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – *Graph and tree search strategies*; I.5.2 [**Pattern Recognition**]: Design Methodology – *Classifier Design and Evaluation*.

main categories: *univariate* decision trees (also called orthogonal), and *multivariate* decision trees. While univariate decision trees test only one attribute per node, the multivariate ones test more attributes.

Most of the decision tree induction algorithms are based on the following greedy algorithm:

BASIC TREE INDUCTION ALGORITHM

- *If all the training examples at the current node t belong to category C, create a leaf node with the class C.*
- *Otherwise, score each one of the sets of possible splits S, using a quality measure.*
- *Choose the best split s^* as the test at the current node.*
- *Create as many child nodes as there are distinct outcomes of s^*. Label edges between the parent and child nodes with outcomes of s^*, and partition the training data using s^* into the child nodes.*
- *A child node t is said to be pure if all the training samples at t belong to the same class. Repeat the previous steps on all impure child nodes.* [4]

One of the most early, and most famous decision tree induction algorithms is the ID3, invented by Quinlan. ID3 uses the *information gain* as quality measure, to decide at which attribute should split the data. Newer versions of ID3, C4.5, and C5 use the *gain ratio*, to qualify an attribute. Other inducers use other methods to calculate the measure of goodness. For example, the CART algorithm, invented by Briemann, uses the *Gini index*. Other tree induction algorithms include: CN2, a multivariate tree inducer, SPRING and SLIQ developed to handle large datasets, OC1, an oblique tree classifier, and others.

After the tree is built, it is usually too *overfit* to be used in classification. This happens if there are too few representative instances in the training data to 'produce' a true target function, or when there is noise in the training data. Growing the tree in this case still all the training instances are 'consumed' can lead to the effect called overfitting. There are two main methods to combat this situation: to stop the tree growth in the induction phase, or to *prune* back the tree. The second method means elimination of some nodes that seem to not have sufficient evidence.

Another way to build a decision tree is to use an evolutionary method. First an initial tree population is generated randomly. Every individual from the population is evaluated, and some of them are selected and are given the chance to reproduce. In a pure generational model offspring replace the parent generation. The process continues until a termination criterion is satisfied. This is usually connected with the maximum number of generations. Koza [5] was one of the firsts, who applied genetic programming to evolve decision trees. Koza used LISP strings to encode the decision trees into chromosomes. There are direct methods, in which the genetic operators operate right on the decision trees [3].

Evolutionary methods can be combined with other powerful heuristics to produce better results. A successful trial for this is presented in [1].

2. An Alternative Encoding Scheme for Evolutionary Algorithms: Multi-Expression Programming

Multi-Expression Programming (MEP), as introduced in [2], adds an extra step of parallelism to the evolutionary algorithm using a special coding technique. MEP was successfully used in several areas like symbolic regression, evolution of game strategies, or in NP-complete problems like the traveling salesman problem.

MEP uses linear chromosome encoding. Genes of variable length build up every chromosome. Each gene encodes a terminal or a functional symbol. A gene encoding a function contains pointer towards the function arguments. Function parameters are always on a 'lower' level in the chromosome than the function itself. Let us consider the following example:

$1 : a$
$2 : b$
$3 : +\ 1, 2$
$4 : c$
$5 : d$
$6 : +\ 4, 5$
$7 : *\ 3, 6$

This chromosome has seven phenotypic transcriptions, namely:

$E1 : a$
$E2 : b$
$E3 : a + b$
$E4 : c$
$E5 : d$
$E6 : c + d$
$E7 : (a + b) * (c + d)$

Generally each chromosome encodes a number of expressions equal to the number of genes it contains. This is the source of a very strong implicit parallelism.

One might ask which phenotypic transcription should be used when the fitness of a chromosome has to be computed? A possible method is to select the expression, which gives the best fitness. Another way to solve this problem is to let several expressions represent a given chromosome. According to [2] this gives supplementary power to the method.

MEP uses the following evolutionary algorithm:

MEP ALGORITHM
begin
    Generate Initial Population;
    $t = 0$;
    Evaluate_Individuals;
    while not Termination_Condition do

Elitism;
Selection;
Recombination;
Mutation;
Evaluate_Individuals;
endwhile
end

For more details regarding MEP, see [2].

3. MEP-Based Decision Tree Induction

An evolutionary classifier system, called MEP-Based Decision Tree Induction (MEPDTI) is proposed. MEPDTI uses the MEP technique to represent the individuals. Classifier systems are used to classify instances from a given data set. Each instance has a set of attributes. Based on the training data set a classifier system builds an internal, and usually compressed representation of the data. This representation is not necessary a decision tree. It may consist from a set of *decision rules*, or some hybrid solution, as in our case.

3.1. **MEPDTI Classes.** Instances have to be classified into classes representing meaningful categories. In our model classes are described as simple character strings.

3.2. **MEPDTI Attributes.** Each instance is characterized by a set of attributes. In MEPDTI attributes can be of three different forms: *nominal*, *discrete*, and *continuous*.

Nominal attributes are symbolic types, which cannot be ordered. They are characterized with a set of possible values they can take. An example for nominal attribute could be *colour*, with the set of possible values {*red, green, blue*}. Boolean attributes are considered as nominal attributes.

Discrete attributes are characterized by predefined sets of numercal values. An example for a discrete attribute is *age*. This attribute takes values from the set {*0, 1, 2, . . . , 130*}

Continuous attributes are ordered attributes, without a set of predefined values. Optionally a lower and an upper bound can be specified for them. An example for such attribute is *temperature*, which takes values from $(0, +\infty)$ (in Kelvin degrees).

3.3. **MEPDTI Instances.** An instance is a vector consisting of possible values of the different attributes, characterizing instances.

Example. Consider the problem of deciding whether a financial organization should or should not offer loan to somebody. The decision relies upon:

- the year income of the applicant (discrete attribute),
- the age of the applicant (discrete attribute),
- criminal records (nominal attribute).

In this case a possible instance representing a person could be:

$$(10000, 47, no).$$

This means, that the applicant has a 10000 income per year, is 47 years old, and has no criminal records.

Usually instances are divided in two categories: *training instances* and *test instances*. While training instances are used to build the decision tree, test instances are used to test the resulting decision trees. Regardless to its type, each instance has associated the correct classification.

3.4. **MEPDTI Representation.** In order to evolve classifiers, the decision mechanism has to be encoded into chromosomes. We use MEP-like chromosomes to represent individuals in MEPDTI. A MEPDTI individual has the following structure:

1:	$Class_1$
2:	$Class_2$
...	
n:	$Class_n$
$n+1$*:*	*if* $(condition_1^{n+1})$ *then jump to* jp_1^{n+1},
	if $(condition_2^{n+1})$ *then jump to* jp_2^{n+1},
	...,
	else jump to jp_k^{n+1};
...	
$n+m$*:*	*if* $(condition_1^{n+m}$ *then jump to* jp_1^{n+m}
	if $(condition_2^{n+m}$ *then jump to* jp_2^{n+m},
	...,
	else jump to jp_1^{n+m};

Let us take a closer look to this general form of the MEPDTI chromosome. It is made up by *genes* (a line in the chromosome). In each line, the first number is the gene identification number. This number identifies a gene in the chromosome (but it does not belong to the chromosome). They will be referred as *jump points, entry points* or *nodes*. There are two types of genes: *terminal*, and *non-terminal* genes. Terminal genes are classes and they occupy the first n loci in the chromosome.

Non-final genes are a little bit trickier. They are made up by decision rules. The general form of a decision rule is:

if (condition) then (action).

The *condition* part a tests an instance with respect to an attribute. The general form of a condition is:

(attribute of instance) (operator) (possible value of the attribute)

Depending on the type of the attribute, different operators have been defined:

- for nominal attributes the unique operator is " = ", which test the equality of two nominal values;
- for discrete and continuous attributes the following operators are used: " < " (less), " <= " (less or equal), " > " (greater), " >= " (greater or equal), " = " (equal).

Let us suppose that we want to test the 'year income' attribute at an instance (10000, 47, no). Supposing that the 'year income' attribute takes values from set $\{0, 1, 2, \ldots\}$, a possible condition would be:

$$10000 <= 7329,$$

where 7329 is a possible value of the attribute 'year income', and " <= " is a relational operator. Because this condition evaluates to false, no action is made.

The *action* part of a decision rule, indicates a jump at one of the jump points of the chromosome. The jump point must be smaller than the actual identification number of the gene. This restriction is introduced to avoid recurrent jumps and to obtain only syntactically correct individuals by recombination.

If a condition satisfied a jump is made. But what if the condition is not satisfied? Then the next decision rule from the gene is considered. It could happen that none of the conditions is satisfied. To handle this situation, we introduced an *else* branch, which guaranties that the gene can be leaved in any circumstances.

3.5. **Classification.** The primary job of a chromosome C is to classify instances from different data sets. The classification accuracy of a chromosome is strictly connected whith its fitness. The better a chromosome classifies a data set, the higher its fitness is. For classifying an instance, we start at some 'entry' point of the chromosome, apply the decision rules found there, jump if needed, and continue this until one of the final genes (classes) are reached. The following pseudocode gives the classification algorithm:

CLASSIFICATION ALGORITHM

```
function classify(instance, entry point)
{
        jump to gene denoted by the entry point;
        while (this is not a final gene) do
        {
                take the first rule from gene ;
                while (rule evaluates to false)
                {
                        take next rule;
                }
                if there are no more rules, then jump to node contained by the
'else' branch;
                else jump to node contained by the rule, which evaluated to true;
        }
        return (the class belonging to the final gene)
}
```

Note that due to the MEP structure, the classification highly depends on the chosen entry point. This is because starting from different entry points result in different set of decision rules.

3.6. **Calculating Fitness.** The fitness of an individual in MEPDTI is the measure of how well this individual can classify a given set. So, the fitness is given by the following formula:

$$fitness = \frac{number\ of\ well\ classified\ instances\ from\ set\ S}{total\ number\ of\ instances\ from\ set\ S}$$

An important question arises: where to start the classification, which should be the entry point? To explore the massive parallelism given by the MEP encoding scheme, when the fitness of an individual is calculated ALL possible entry points are used, and the one, which leads to the most correctly classified instances gives the '*number of well classified instances from set S*' used in the above formula.

FITNESS ASSIGNMENT ALGORITHM

```
function fitness(chromosome, set of instances)
{
        max = 0;
        for (every instance x from the set S)
        {
                nr=0;
                for(every entry point ep from the chromosome C)
                {
                        if (classify(x, ep)==x.class) then nr++;
                }
                if (max < nr) max = nr;
        }
        return(max/number of instances in the set);
}
```

Remark. *x.class* denotes the correct classification of instance x, as it is given in the training data set.

3.7. **Variation Operators.**

3.7.1. *Recombination.* Recombination ensures the mixture of genes. There are three types of recombination operators in MEPDTI: one-point crossover, two-points crossover, and uniform crossover. All of them affect only the outer part of the chromosome: none of the genes are broken during recombination operations.

3.7.2. *Mutation.* Mutation slightly perturbs a chromosome. It affects only the inner part of the chromosome, (i.e. the genes, and the sub-gene structures). In our implementation the variation operator can take one of more of the following actions:

- change either of operands from a condition
- change the operator from a condition
- change the jump point from a decision rule, or the else branch
- insert or delete decision rules from/into the gene
- change the attribute on which the tests are made from the gene

3.8. **MEPDTI Algorithm.** We are ready now to present the MEPDTI evolutionary algorithm.

MEPDTI ALGORITHM

```
Function MEPDTI(max_epochs)
{
        t = 0;
        Generate a randomly initialized population;
        While (t < max_epochs)
        {
                Calculate the fitness of every individual from the population;
                Sort by fitness the individuals from population;
                While(there are unoccupied positions in the new population)
                {
                        Select two individuals from the population, based on their
                        fitness using some selection technique;
                        Recombine the two selected chromosomes with a specified
                        probability, using some recombination technique;
                        Apply the mutation operator on the resulting two offsprings
                        with a specified probability, using some of the mutation types;
                        Put the offsprings into the new population;
                }
                Replace the population with the new one;
                t++;
        }
        Sort by fitness the individuals from population;
        Return (the best individual);
}
```

3.9. **Classification Revisited.** Let us suppose that the evolutionary process ends and we have the best chromosome. One might ask how does this chromosome classify an unknown instance. We reevaluate the chromosome on the training set, and find the entry point, which leads to the most correctly classified instances. This entry point is used then to classify the unknown instance.

3.10. **MEPDTI Complexity and Strong Parallelism.** The strong parallelism hold by MEP has a serious drawback: the evaluation must be done on each possible entry point. This increases the classification complexity of a chromosome

(classifying a single instance) in the worst case from $O(n)$ to $O(n^2)$, where n is the number of genes in the given chromosome.

MEPDTI time complexity is $O(m \cdot q \cdot e \cdot n^2)$, where m is the size of the training data, q is the number of chromosomes from the population, and e is the number of generations.

3.11. **Example.** In this section the example of Quinlan [6] is considered. The problem is to decide whether to play or not golf, depending on the weather conditions. There are two classes in this problem {play, don't play}, and four attributes:

- outlook, with possible values: {sunny, overcast, rain}
- temperature, which is continuous
- humidity, also continuous
- windy, with possible values: {true, false}

In the data set there are 14 instances:

```
overcast, 64, 65, true, play
overcast, 72, 90, true, play
overcast, 81, 75, false, play
overcast, 83, 78, false, play
rain, 65, 70, true, don't play
rain, 68, 80, false, play
rain, 70, 96, false, play
rain, 71, 80, true, don't play
rain, 75, 80, false, play
sunny, 69, 70, false, play
sunny, 72, 95, false, don't play
sunny, 75, 70, true, play
sunny, 80, 90, true, don't play
sunny, 85, 85, false, don't play
```

A chromosome evolved by MEPDTI in 250 generations which classifies 100% the above data set is:

```
  0::<<Play>>
  1::<<Don't Play>>
  2::[humidity]::if inst>70.00 then jump to:  0;; if inst>=85.00 then
jump to:  0;; else jump to:  0;;
  3::[windy]::if inst=true then jump to:  1;; else jump to:  0;;
  4::[humidity]::if inst<=70.00 then jump to:  0;; else jump to:  1;;
  5::[outlook]::if inst=sunny then jump to:  4;; if inst=overcast then
jump to:  0;; else jump to:  3;;
```

The best entry point found for test set is: 5. If you look closer to the chromosome, you may see that there is an unused gene, labeled with 2, which can be pruned. After pruning the tree has the following form:

```
0::<<Play>>
1::<<Don't Play>>
```

```
       2::[windy]::if inst=true then jump to:  1;; else jump to:  0;;
       3::[humidity]::if inst<=70.00 then jump to:  0;; else jump to:
1;;
       4::[outlook]::if inst=sunny then jump to:  3;; if inst=overcast
then jump to:  0;; else jump to:  2;;
```

3.12. **Handling Continuous and Missing-Valued Attributes. Pruning.** Handling continuous and missing-valued attributes and tree pruning are important topics in the decision tree induction field. In MEPDTI we used some simple techniques to solve these problems.

Continuous attributes are used without any modification, as they are in the training or test data files. They are mainly used in the random initialization of the chromosomes, and the in the mutation phase.

Before the evolutionary process starts, both data and test files are parsed to eliminate the missing-valued attributes. A simple heuristic is used: if a ,,?" is found in an instance x (meaning that there is missing value), it is replaced by the most common value from instances which are classified in the same class as x.

When evolutionary process ends, MEPDTI returns the most successful chromosome, and the most successful entry point of this chromosome. Usually there are some jump points that are not used (starting from the most successful entry point). These corresponding genes are eliminated using a pruning procedure. This procedure parses the chromosome starting at a given entry point, and deletes those genes which are not used in the classification process.

4. Numerical Experiments

MEPDTI produces classifiers whose accuracy is similar with those produced by the classical decision tree inducers, like CN2 or C4.5. For test purposes data from the UCI Repository Of Machine Learning Databases and Domain Theories [7] have been used. The following table presents the results of the MEPDTI. The results of CN2, C4.5, and BGP, an evolutionary method were taken from [3].

The following settings were used:

- population size: 1000
- number of generations: 250
- variable length chromosome usage
- mutation probability: 0.9 (!)
- crossover probability: 0.9
- tournament selection, with tournament size = 4
- no fitness remapping
- two point, or univariate crossover
- strong mutation, meaning that all mutation types described in the previous part of this article are used
- elitism: 1-5%

Data set name	Number of classes	Number of attributes	Classification ratio	Best entry point's position (after prunning)	Classification ratio with the best entry point	Number of genes in the best chromosme (after prunning)	CN2 classification ratio	C4.5 classification ratio	BGP classification ratio
crx	2	15	0.83	last	0.83	3	?	?	?
hypo	5	29	0.95	last	0.95	1	?	?	?
ionosphere	2	34	0.91	last	0.91	7	0.92	0.93	0.89
iris	3	4	0.95	last	0.95	3	0.94	0.94	0.94
monk-1	2	6	0.85	last	0.85	1	1.00	1.00	0.99
monk-2	2	6	0.64	last	0.64	3	0.62	0.63	0.68
monk-3	2	6	0.97	last	0.97	3	0.90	0.96	0.97
pima	2	8	0.77	last	0.77	3	0.72	0.73	0.72
tic-tac-toe	2	9	0.75	last	0.75	3	?	?	?
votes	2	16	0.97	last	0.97	5	?	?	?
soybean	19	35	0.31	last	0.31	2	?	?	?

FIGURE 1. Table showing the results of MEPTDTI

The first remark to the test runs is that although all possible entry points are used for classification, usually the last entry point has the biggest success. However, there were cases when the most successful entry point was an intermediate one.

Another observation is the increased importance of the mutation operator against the crossover. On some data sets the results were the same even if the crossover was given zero probability. The total pass over of the crossover operator needs more investigations.

4.1. **Future Work.** In our approach, only one attribute is tested at each gene. Thus a MEPDTI chromosome corresponds to an orthogonal decision tree. In order to get rid of the drawbacks of univariate decision trees, we intend in the next version to evolve multivariate classifiers.

References

[1] Bala J., Huang J., Vafaie H., DeJong K., Wechsler H., *Hybrid Learning Using Genetic Algorithms and Decision Trees for Pattern Classification.* IJCAI Conference, Montreal, 19-25, 1995.

[2] Oltean M., Dumitrescu D., *Multi Expression Programming*, submitted to Journal of Genetic Programming, 2003.

[3] Rouwhost S.E., Engelbrecht A.P., *Searching the Forest: Using Decision Trees as Building Blocks for Evolutionary Search in Classification Databases.*Proc. Congress on Evolutionary Computation (CEC-2000), 633-638. La Jolla, CA, USA, 2000.

[4] Murthy, S.K., Kasif, S., Salzberg, S., *A System for Induction of Oblique Decision Trees*, Journal of Artificial Intelligence Research 2, 1-32, 1994.

[5] Koza, J., *Genetic Programming: On The Programming of Computers by Means of Natural Selection,* MIT Press, Cambridge, MA, 1992.

[6] Quinlan, R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Meteo, CA, 1993.

[7] UCI Machine Learning Databases and Domain Theories, *ftp.ics.uci.edu: pub/machine-learning-databases*

Department of Computer Science, Babes-Bolyai University, Cluj
E-mail address: jooandras@ms.sapientia.ro
E-mail address: ddumitr@nessie.cs.ubbcluj.ro