Generalized Decision Trees Built With Evolutionary Techniques

D. Dumitrescu

Department of Computer Science

Babes-Bolyai University, Cluj,

ROMANIA

JOÓ András

Department of Mathematics and Information Sciences

Sapientia University, Targu Mures,

ROMANIA

Abstract: This paper proposes a new decision tree type: Generalized Decision Tree (GDT). These trees contain completely generalized tests in their internal nodes, hence they describe much better the data than their traditional counterparts. An evolutionary method to build such decision trees is presented.

Keywords: decision tree, evolutionary algorithm, genetic programming, multi-expression programming

D. Dumitrescu obtained his PhD in 1990 from University of Cluj, Romania, in Mathematics and Computer Science. During the period 1998-2000 he was invited professor and researcher at the University of Pisa , Italy. Presently he is full professor in the Department of Computer Science, University Babes- Bolyai, Cluj. His main research fields are Natural Computing, Evolutionary Computing, Computational Intelligence and Fuzzy Systems.

Joo Andras obtained his Mathematics and Information Sciences degree from Petru Maior University, Târgu Mureş, Romania, in 2002. From October 2002 he is a assistant at the Sapientia University, Târgu Mureş. In 2003 he obtained a MSC degree in Intelligent Systems from the Babes-Bolyai University, Cluj. Currently he is a PHD student. His main research fields are evolutionary techniques and decision mechanisms.

1. Introduction

Decision tree building is an intensively studied field of the machine learning. The research has been focused for long time mainly on the greedy induction of the univariate decision trees. Famous algorithms of this kind are the ID3 and its descendant C4.5. [10]. In both algorithms the internal nodes test a given instance against a single attribute. This kind of test splits the search space parallel with the axis of the search space. Sometimes this is not the most adequate approach and one or more of the tests have to be repeated, in order to obtain a good classification ratio. This leads to replication of subtrees.

The univariate decision tree inducers usually use measures derived from information theory to decide on which attribute to split on. The main task is to build the smallest decision tree still consistent with the training data. This problem is known to be NP hard.

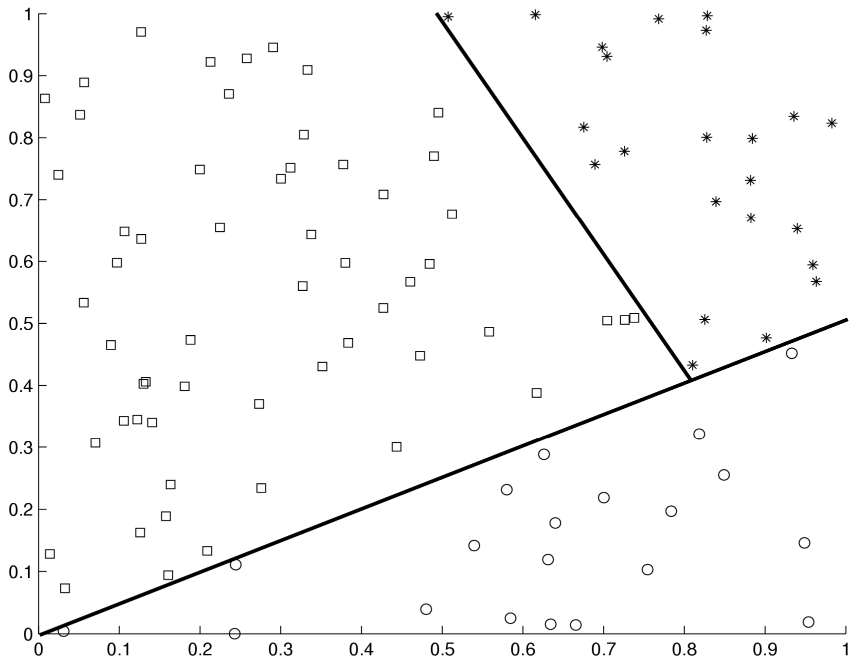


Figure 1: This Problem Needs Oblique Splits.

Also much research has been done on the induction of oblique decision trees. These contain linear combinations of the instance values in their internal nodes, hence they split the search space in an oblique way (See Figure 1). A well known algorithm is the OC1([10]). In these the main problem is to find the coefficients of the linear combinations. This problem is also NP hard.

Standard decision tree types split the search space linearly. There are problems where the perfect split is not linear at all. A well known example is the two spirals problem (See Figure 2). A further step in the generalization are the non-linear multivariate decision trees. The test in these trees usually split the search space along some second degree curved hyper surface.[4]

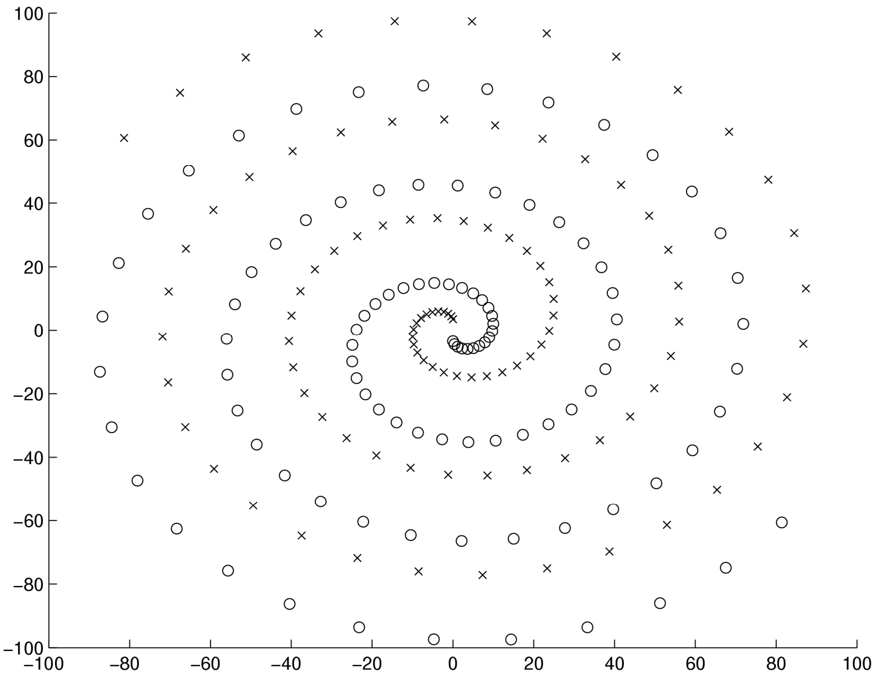


Figure 2. This Spiral Problem Needs non-linear Splits.

The concept of generalized decision tree (GDT) is proposed. A GDT contains in its internal nodes general tests which involve one or more attributes. GDTs give a more compact representation of data than their orthogonal or oblique counterparts, although their interpretation might be difficult. To describe such a general test we use trees, so the final decision tree is a tree which contains internal trees in its non-terminal nodes.

The paper is organized as follows: section 2 explains the need for GDTs. Section 3 describes the basis of the used encoding, MEP. Section 4 extends MEP codification scheme to fit GDT requirements. The next three sections describes the details of the used evolutionary technique: search operators, fitness measuring, and the evolutionary algorithm itself. Finally some preliminary results are presented and further research directions are drafted.

2. Why Generalized Decision Trees?

The reason for using generalized tests in the internal nodes of the decision trees is motivated by the following observation: the more precise the tests in the internal nodes are, the smaller the tree becomes. Our aim is to find the most suitable tests for the internal nodes of the tree. The complexity of these tests can vary from a simple inequality to very complex expressions. To represent the internal tests, we have chosen trees, which have in their internal nodes predefined operators (logical, relational, trigonometrical, etc functions), and in their leaves different values which are derived from the input data. Such of expression evaluates to a numerical value. If this value is non-null, we say that the test is satisfied, otherwise it is not. We call these tests generalized tests.

The generalized decision trees have the big advantage that cover all the classic decision tree types developed until now. On one hand, if the general tests consist of testing single attributes then we deal with an univariate decision tree. On the other hand, if the whole tree is a single big test then we deal with genetic programming expressions (see [6]).

However, there are some drawbacks: the more complicated tests are in the internal nodes, the the less comprehensible are by humans. This, and the increased computational effort needed to find the exact general tests are the price that usually have to be paid for the compactness. Also, the more compact the tree is, the greater the probability that over-fits the data [2].

One might ask how to find the exact expressions for the internal nodes. Going through all the possible expressions is infeasible, so we need some heuristic method. The evolutionary method was selected since it has proved its capabilities in similar situations. ([1,5,6,9,11])

First a number of decision trees are generated randomly. Those who perform well (meaning that their classification capabilities are good) are allowed to mate and have children. Bad individuals are removed from population, they are replaced by the offspring of the parents selected in the previous steps. This procedure is continued until the best individual reaches a given level of goodness, or a given number of generations have passed. Then the best individual is used for classification tasks.

To do the evolution, first we need an adequate codification of the possible solutions, we need evolutionary operators, some goodness measure, and a concrete evolutionary algorithm.

3. MEP Encoding

Encoding is a method which converts a possible solution, called *phenotype,* into a structure (*genotype*) that will undergo adaptation during the evolutionary process . In our case the genotype will consist of a single *chromosome*, built up by *genes* (see details later).

MEP encoding (Multi Expression Programming, see [9]) allows to rewrite a given tree in a linear fashion. The first elements of the MEP structure are atomic. The rest are either atomic or compound elements. Compound elements contain functions that take as arguments the indexes of other elements. To avoid recursion the index of a compound node must be greater than any of the parameters' indexes it contains.

Let us give an example for the MEP codification. Consider chromosome *C* defined below:

1: a
2: b
3: + 1, 2
4: sqr 3
5: c
6: - 4, 5

This linear structure encodes the tree presented on Figure 3.

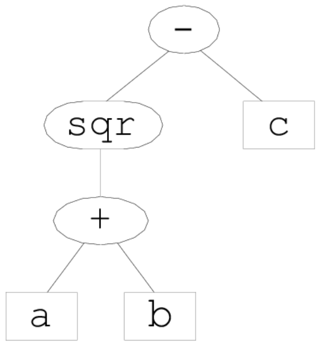


Figure 3. The Tree Corresponding to Chromosome *C*.

The interesting thing about this encoding type is that the evaluation depends on the entry point, ie. the point the evaluation starts from. For example if we take the entry point 6, we will get the expression $(a+b)^2$. But if the evaluation starts at point 3, we will get a simpler expression: $(a+b)$. This property is very useful in evolutionary algorithms, since a genome encoded by a MEP structure will have multiple phenotypic transcriptions. To find the best phenotypic transcription all of the entry points have to be checked. This introduces an extra computational cost.

4. GDT Encoding

The GDT encoding scheme is a modified version of the MEP codification designed to hold generalized decision trees. It contains on its first positions the possible classes an instance can be classified. The rest of the elements are tests which test the instance regarding to one or more attributes. The general form of a GDT structure containing *n* classes and *m* tests is presented in Figure 4. The result of every test is either *true* (non-zero) or *false* (zero), hence the corresponding decision tree is a binary. For simplicity we have restricted the usage of atomic elements (classes) by means that they must occupy the first *n* positions of the chromosome.

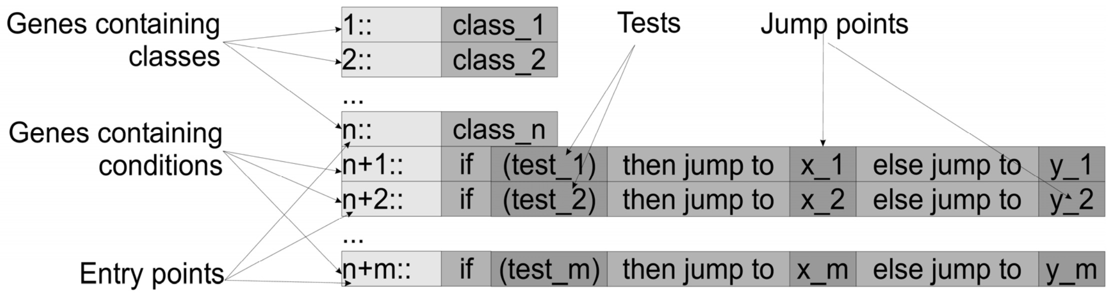

Genes containing classes
Genes containing conditions
Entry points
Tests
Jump points
1:: class_1
2:: class_2
...
n:: class_n
n+1:: if (test_1) then jump to x_1 else jump to y_1
n+2:: if (test_2) then jump to x_2 else jump to y_2
...
n+m:: if (test_m) then jump to x_m else jump to y_m

Figure 4: Logical view of a GDT chromosome. A chromosome is made up by genes: class-genes in front, followed by condition-genes. Every gene has a number, a so called entry point, which identifies the gene. A condition contains a test (a tree) and two jump points: the first one is used if the test evaluates to true, the second is used if the test evaluates to false.

Every *test_i, i=1...m* is a tree having in its internal nodes operators or functions, and in its leaves values derived from instance or attribute values. We call these tests *internal trees*. An example for such of internal tree written in infixed Polish form is the following:

((ABS(temperature))/(COS(0.85841)))

Here *temperature* is an attribute, and 0.85841 is a randomly generated value from a possible value of an attribute. It is permitted to an operator to have as parameters both exact values or attributes. Thus the following expressions are also correct:

(0.819752&&0.870315)

or

(LN(temperature/humidity))

In the internal trees the following operators and functions are used:

- logical operators: `and, or, not`
- relational operators: `<, >, <=, >=, ==, ! =`
- arithmetical operators: `+, -, *, /`
- trigonometric functions: `SIN, COS`
- exponential and logarithm functions: `EXP, LN`
- absolute value: `ABS`
- square and square root functions: `SQR, SQRT`

Some of the operators and functions have been rewritten to assure that the program will not crash in the case of illegal or unusual operands. Thus:

- the logical and the relational operators return 0 or 1, depending whether the condition is satisfied or not
- the natural logarithm in fact is the natural logarithm of the absolute value of the operand, ie. *ln*(*x*) is mapped to *ln*(*abs*(*x*))
- the square root is the square root of the absolute value of the operand, ie. *sqrt*(*x*) is mapped to *sqrt*(*abs*(*x*))
- the division is rewritten so that if the divisor is null, it is replaced by a very small positive value

It has to be mentioned that because the tree corresponding to a GDT chromosome is binary, and the function set used in tests doesn't contain loops, if-else structures, neither the assignment operator (=), GDTI (Generalized Decision Tree Inducer) is not a fully implementation of a generalized decision tree.

5. Calculating the Fitness

The fitness of an individual must reflect its classification ability. Hence it is calculated as the percent of well classified instances from the test set. Since a GDT chromosome has multiple phenotypic transcriptions, an individual has multiple fitness values. The selection operator will work only with the best fitness value the individual got. In the following we give the pseudocode for calculating the fitness of a GDT chromosome:

```
function CalculateFitness(chromosome, testSet)
fitness = 0
for every non-class entry point e from chromosome
      sum = 0
      for every instance ins from testSet
            i = e
            while (i is not a class)
                  if (test_i.eval(ins) == true) i = x_i
                  else i = y_i
            end
            if (class_i == ins.class) sum++
      end
      sum /= testSet.length
      if (fitness < sum) fitness = sum
end
return fitness
```

Now let us explain the steps of fitness calculation. First we have to check all the non-class entry points the chromosome has. We pick instances from the test set, and propagate through the chromosome starting from entry point *i*=*e*. Depending on whether the test at entry point *i* (*test_i*) is true or false, we jump to *x_i* or *y_i*. When an entry point containing a class has been reached, it is compared with the correct classification of the instance (*ins.class*). If they match, a counter (*sum*) is increased. After all the instances has been tested, the counter is divided by the number of instances from the test set (*testSet.length*), to find the ratio of well classified instances for entry point *e*. Finally the global fitness (*fitness*) is compared with the fitness of the entry point (stored now in *sum*), and it is updated if necessary.

6. Search Operators

In the following we will describe the details of the used search operators.

Mutation

Mutation makes little modification in the involved chromosome. Mutation usually helps to maintain the diversity of the population by inserting individuals which could not have been "made" just by crossover. There are several levels where mutation can be applied. The level on wich actually the mutation will occur is selected in a random manner.

- Chromosome level: one gene is inserted/deleted randomly, or the mutation is propagated to a lower level.
- Gene level: has meaning only in the case of non-class genes. The condition is replaced randomly or the mutation is propagated to a lower level.
- Condition level: the contained tree is replaced, one of the jump points is altered or the mutation is propagated to a lower level.
- Tree level: a subtree is selected and replaced randomly.

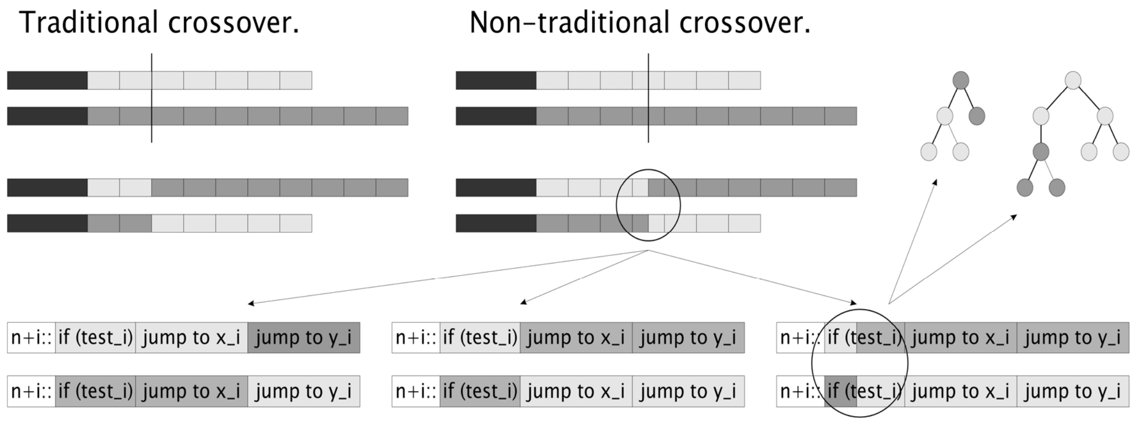
Crossover

Crossover takes usually two individuals called parents, and produces one or more offspring. Offspring inherit parts of their parents' genetic material. The crossover operator also works on different levels:

- Chromosome level. Since chromosomes are linear structures the typical one- or two-point or uniform crossover can be used. The crossover point must point above the class-genes. In contrast with the usual genetic crossover, where the genes were atomic elements, a GDT gene can be split by the crossover operator. If the crossover point points between to genes, the traditional crossover is used. If it points to a gene, then the corresponding genes are split. This is achieved by calling a lower level crossover method.

- Gene level. The condition-level crossover is called.
- Condition level. Condition is also a linear structure having three elements: the tree and the two jump points. One-point crossover is used at this level. If the crossover point points to the tree, then a lower level (tree level) crossover is made.
- Tree level. Tree needs specialized crossover operator, so we used one borrowed from genetic programming: two nodes are selected randomly on both trees and the subtrees defined by these are swapped.

The different crossover types are illustrated on Figure 5.



Traditional crossover.
Non-traditional crossover.
n+i:: if (test_i) jump to x_i jump to y_i
n+i:: if (test_i) jump to x_i jump to y_i
n+i:: if (test_i) jump to x_i jump to y_i
n+i:: if (test_i) jump to x_i jump to y_i
n+i:: if (test_i) jump to x_i jump to y_i
n+i:: if (test_i) jump to x_i jump to y_i

Figure 5. Different crossover types in GDTI. The Figure illustrates the different crossover types available at different levels. The upper bars on top represent the chromosomes which will suffer crossover, every slice being a gene. The dark parts at the fronts of the bars are the class-genes, which do not enter the game. The upper-left part of the Figure presents the traditional one-point crossover. The right part of the Figure shows a non-orthodox crossover, where a gene is split in two. There are three different possible results, as shown. The right-most is also unusual, since the ith tests are split. This is a tree crossover, where two nodes are selected randomly on both trees and the subtrees defined by these are swapped.

Selection

Selection is used to select individuals for some events like mutation, crossover, or elimination from the current population. GDTI uses tournament selection as selection for crossover: two or more individuals are selected randomly from population and the one with the greatest fitness wins the "tournament". The more individuals enter the contest, the higher the selection pressure is. For mutation we use random selection, meaning that every individual has the same probability for being selected.

Steady-state selection is used as population replacement strategy. This enables the survival of the fits, and the competition of old and new individuals.

6. Evolution

Having all the operators necessary to evolve decision trees we can now give the evolutionary algorithm.

```
ALGORITHM GDTI
time = 0
Initialize(population)
while(time < generations)
CalculateFitness(population)
matingPool = TournamentSelection(population)
offspring = Crossover(matingPool)
offspring = Mutate(offspring)
UpdatePopulation(population, offspring)
time ++
end
```

Let us explain the presented algorithm. First the population is initialized with random data derived from the training set. Then follows the evolutionary loop, which lasts until the time variable has reached a previously fixed limit (the number of generations). The evolutionary loop starts with calculating the fitness of the individuals. The individuals with the best fitness are selected using the tournament selection scheme into an intermediate location, called the mating pool. Offspring are produced by pairwise crossover of parents chosen randomly from the mating pool. Mutation is applied to offspring, then the offspring replace the bad individuals from the original population. The evolutionary loop ends with the increment of the time variable.

7. Results

Data Preparation

The training and test data were taken from the UCI Machine Learning Repository1 . The data files have been parsed and the numeric values have been normalized to mean 0 and deviation 1.

In the current version of GDTI the symbolic attributes are left out from the game, ie. they are not used in the internal tests.

To handle the missing values we have chosen the fastest (but not necessarily the best) method: if an instance has a missing value at some attribute then the most common value among the other instances at same attribute is put in the place of it. If in the training set all the values are missing at some attribute then the respective attribute is removed.

Results

The results are quite positive, although there are datasets where we had expected better classification ratios. Table 1 contains a comparation with some standard algorithms (C4.5, CN2, BGP). (MEPDTI is an evolutionary, orthogonal decision tree inducer([5])).

Data set	Number of classes	Number of attributes	GDTI	MEPDTI	CN2	C4.5	BGP
breast-cancer	2	10	0.979	n.a.	n.a.	n.a.	n.a.
glass	7	9	0.767	n.a.	n.a.	n.a.	n.a.
ionosphere	2	33	0.957	0.910	0.920	0.930	0.890
iris	3	4	0.987	0.950	0.940	0.940	0.940
monk1	2	6	1.000	0.850	1.000	1.000	0.990
monk2	2	6	0.816	0.640	0.620	0.630	0.680
monk3	2	6	0.959	0.970	0.900	0.960	0.970
pima	2	8	0.807	0.770	0.720	0.730	0.720
spirals	2	2	0.823	n.a.	n.a.	n.a.	n.a.
tic-tac-toe	2	9	0.863	0.750	n.a.	n.a.	n.a.
wine	3	13	0.977	n.a.	n.a.	n.a.	n.a.

Table 1. Results obtained with the Genarailzed Decision Tree Inducer (GDTI).

1ftp.ics.uci.edu:pub/machine-learning-databases

8. Further Work

Among of our plans are the fine-tuning the algorithm to give a better solution for the highly non-linear problems, like the spirals problem. Also, we are planning to investigate the effect of the used search operators, and if there is true need for the strong parallelism given by the MEP encoding.

REFERENCES

1. BALA J., HUANG J., VAFAIE H., DEJONG K., WECHSLER H., **Learning Using Genetic Algorithms and Decision Trees for Pattern Classification. IJCAI Conference**, Montreal, 1995.
2. BENETT K.P, CRISTIANINI N., SHAWE-TAYLOR J., WU D., **Enlarging the Margins in Perceptron Decision Trees**, 1999
3. BREIMAN L., FREIDMAN J.H, OLSHEN R.A., STONE CH.J, **Classification and Regression Trees**, Chapman&Hall, 1984
4. ITTNER A., SCHLOSSER M., **Non-Linear Decision Trees - NDT**, Proceedings of 13 International Conference on Machine Learning - IML96, Lorenza Saitta, 1996
5. JOÓ A., DUMITRESCU D., **Evolving Orthogonal Decision Trees**, Studia Universitatis Babes Bolyai, Cluj, 2003
6. KOZA J., **Genetic Programming: On The Programming of Computers by Means of Natural Selection**, MIT Press, Cambridge, MA, 1992
7. LLORA X., GARREL J.M, **Evolution of Decision Trees, Proceeding of the 4th Catalan Conference on AI**, ACIA Press, 2001
8. MURTHY S.K, KASIF S., SALZBERG S**. A System for Induction of Oblique Decision Trees**, OC1, Journal of Artificial Intelligence Research 2, 1994
9. OLTEAN M., DUMITRESCU D., **Multi Expression Programming**, submitted to Journal of Genetic Programming, 2003
10. QUINLAN R., **C4.5: Programs for Machine Learning, Morgan Kaufmann**, San Meteo, CA, 1993
11. 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