Coevolution of Data Samples and Classifiers Integrated with Grammatically-based Genetic Programming for Data Classification

Douglas A. Augusto
COPPE/UFRJ
Rio de Janeiro, Brazil
douglas@coc.ufrj.br

Helio J.C. Barbosa
LNCC/MCT
Petrópolis, Brazil
hcbm@lncc.br

Nelson F.F. Ebecken
COPPE/UFRJ
Rio de Janeiro, Brazil
nelson@ntt.ufrj.br

ABSTRACT

The present work treats the data classification task by means of evolutionary computation techniques using three ingredients: genetic programming, competitive coevolution, and context-free grammar.

The robustness and symbolic/interpretative qualities of the genetic programming are employed to construct classification trees via Darwinian evolution. The flexible formal structure of the context-free grammar replaces the standard genetic programming representation and describes a language which encodes trees of varying complexity. Finally, competitive coevolution is used to promote competitions between data samples and classification trees in order to create and sustain an evolutionary arms-race for improved solutions.

Categories and Subject Descriptors

I.5.2 [**Pattern Recognition**]: [Design Methodology, Classifier Design and Evaluation]; H.2.8 [**Database Management**]: [Database Applications, Data mining]

General Terms

Design, Algorithms, Experimentation

Keywords

Genetic Programming, Data Classification, Context-Free Grammar, Competitive Coevolution

1. INTRODUCTION

To classify means to group things based on similarities, giving them a class label. In data classification, a data sample is described by a set of attributes, so the similarity degree among data samples is taken in some way from their attributes. The goal of the data classification task is to extract useful information from a training data set so that it is possible to: (1) accurately and automatically classify unseen but related data samples; (2) obtain a concise and rich human-readable knowledge about the data set.

This work presents an approach based on evolutionary computation that aims to achieve the stated goals of the data classification task, although it concentrates on the aspect of accurate and automatic data classification. The approach consists in the integration of a grammatically-based genetic programming with a competitive coevolution. The genetic programming provides the basic structure of this work, and it establishes the learning method, which is inspired by biological evolution. A context-free grammar representation is used as a formal instrument to specify the language—possibly having complex constraints—of the candidate solutions (classifiers). Finally, the coevolutionary scheme promotes competition between data samples and classifiers in order to boost the evolutionary process.

This paper is organized as follows. Section 2 describes the canonical genetic programming. In Section 3 the standard genetic representation and some of the main alternatives are briefly discussed; also, the formal definition of context-free grammar is shown and then the grammatically-based genetic programming is presented. Section 4 discusses the coevolution of data samples and classifiers and how it integrates with the grammatically-based genetic programming. Some experiments in data classification are performed in Section 5 in order to show the potential of the technique proposed here. Finally, Section 6 outlines some conclusions and directions of future work.

2. GENETIC PROGRAMMING

Genetic programming (GP) [9], is a specialization of the genetic algorithm (GA) [7, 3] designed to evolve computer programs in an arbitrary language—commonly in a symbolic, human-readable language. As a genetic algorithm, GP is based on the Darwinian principle of reproduction and survival of the fittest individuals; the evolutionary dynamics and some genetic operations found in nature, such as recombination (crossover) and mutation, are simulated in genetic programming.

Unlike the canonical binary representation of a GA, GP uses a more suitable manner to represent an individual that encodes a computer program. The data structure known as *tree* is typically employed in GP for the representation of the individuals, not only because of its power to express the dynamics of a program, but also because of its simplicity.

The initial population is composed of individuals each one containing a random tree, that is, a program. Each node in a tree is randomly chosen and it contains either an *operator* (function) or an *operand* (terminal)[1]. What a program can do depends on the domain of application. More precisely, it depends on the language chosen for the problem being solved. For example, in a data classification domain, the language could have conditional statements (e.g. `if-then-else`), logical operators (e.g. `or`, `and`, `not`), and so forth.

The original concept of language in GP, as developed by Koza, was trivial, and it was rudimentary. In the canonical GP, there are two sets that may be freely combined: the function set ($\mathbb{F}$) and the terminal set ($\mathbb{T}$). When combined, these sets define the language of the problem. The set $\mathbb{F}$ is responsible for providing the functions (operators), such as arithmetic operators ($+$, $-$, $\times$, $\div$, $\ldots$), trigonometric functions (sin, cos, ...), and conditionals. By contrast, the set $\mathbb{T}$ provides the operands (functions that do not require arguments), e.g., *constants*, *variables*, and *attributes*.

After the creation of the initial population, each individual is evaluated and its fitness is stored. For data classification this means that an individual's fitness will be proportional to its prediction accuracy, and possibly to another measure like "solution complexity". If the termination criterion is not satisfied—for example, a certain classification accuracy—, then the iterative evolutionary process begins: the fittest individuals are selected for recombination/mutation, and their offspring are inserted in the next generation according to a given policy. The evolutionary loop is repeated until a good solution is found, or until it reaches a maximum number of iterations.

3. GRAMMATICALLY-BASED GP

3.1 Genetic Representation in GP

To ensure that a program will remain feasible throughout the evolutionary process, Koza introduced a requirement called *closure*. It states that all constants, variables, function arguments, and return values must be of the same data type, so that the genetic operations (e.g. crossover and mutation) can take place at any site of a tree structure. The closure requirement is a simple and uniform concept; however, it creates two main problems: (1) since every function/terminal can be combined to one another—and many of such combinations do not make sense in conceptual terms—, there is an unnecessary and undesirable [4, 5] increase in the size of the search space; (2) the closure is hard or even impossible to be satisfied for those problems that require different data types.

Some attempts were developed in order to overcome the issues created by the closure requirement of the canonical GP. Three important approaches will be briefly discussed here.

3.1.1 Constrained Syntactic Structures — CSS

CSS was developed by Koza [9] as an attempt to cope with the closure issue. Basically, problem dependent rules are established to limit which functions and terminals are allowed to stand as argument of another function. The genetic operations (initialization, crossover, and mutation) are modified to ensure that the specified rules are not violated during the evolution, so that every program structure remains valid.

Even though the CSS approach provides a mechanism to avoid the serious problems of the closure requirement, it requires a set of rules to be created for each problem; moreover, instead of the declarative form, CSS requires the rules to be specified in textual form, that is, in human language.

3.1.2 Strongly Typed Genetic Programming — STGP

Montana [10] proposed a generalization of Koza's CSS. His target was the concept of *type*, so that the function specifies exactly the type for each argument and for the value it returns.

In STGP, a variable or constant has an assigned type. For example, the π constant is `float`, whereas the variable x is `integer`. In the same way, a function declares types for both the arguments and the return value.

However, the major contribution of STGP over CSS was the extension of the concept of types, including the *generic function* and the *generic data type*.

A generic function is a function that takes arguments of any type. As a result, a unique function is enough to deal with different argument types. Because the generic function needs to be instantiated during the tree creation process, it requires that the return type can be precisely deduced by specifying the argument types. Once instantiated, the generic function acts like a standard strongly typed function.

A generic data can hold arbitrary types. Montana reports that generic data types reduce the size of the search space and also allow the evolutionary process to evolve generic functions.

One of the weakness of STGP is that there is no direct way to create deep structural relationships, i.e., conceptual constraints. In STGP, a function or terminal is constrained only by its type. But occasionally, even though two functions or terminals have the same type, they have different concepts or requirements; therefore, they should not be mixed together.

3.1.3 Grammatical Evolution — GE

Ryan, O'Neill *et al.* [11] proposed a genetic algorithm called *grammatical evolution* where the representation mechanism is based on context-free grammar[2] (CFG). In doing so, GE can theoretically evolve programs in any language described by a CFG.

A GE's individual uses a variable-length encoding scheme where each gene holds an integer value that will be mapped to previously labeled production rules of a given CFG by the decoding process. For example, consider the grammar shown in Figure 1, which defines a language of mathematical expressions, including sin, cos, arithmetic operations, and the variable x.

Also consider the genome in Figure 2, which has 10 genes with values ranging from 0 to 255 (8-bit number). Since an 8-bit integer is far more than the number of production rules, the modulus operation is needed to decode the genes properly.

The decoding process reads the first gene, 204. There are three productions pointed to by the start symbol `<exp>` (group **I**), so the selected one is the production labeled 0 ($204 \bmod 3 = 0$), that is, `<exp>` $\rightarrow$ `<exp> <op> <exp>`. Next, the second gene is read and its value (143), after the mod-

[1]The process stops when all leaf nodes have a terminal (functions that don't require arguments), so the tree is left feasible.

[2]A formal introduction to context-free grammar can be found in Section 3.2.1.

N =	{<exp>, <op>, <preop>, <var>}				
Σ =	{x, sin, cos, +, −, (,)}				
S =	<exp>				
P =	**I**	<exp>	→	<exp> <op> <exp>	**(0)**
				| <preop> (<exp>)	**(1)**
				| <var>	**(2)**
	II	<op>	→	+	**(0)**
				| −	**(1)**
	III	<preop>	→	sin	**(0)**
				| cos	**(1)**
	IV	<var>	→	x	**(0)**

Figure 1: Grammar describing the functions sin, cos, +, **and** − **over** x.

ulus operation (143 mod 3), results in 2; therefore, the production `<exp>` → `<var>` is picked up. Since there is only one production rule headed by `<var>` (group **IV**), the actual selected production is `<exp>` → `<var>` → x, or just `<exp>` → x. So far the decoded expression is x `<op> <exp>`. The next steps will sequentially produce the expressions $x+$ `<exp>`, $x+$ `<preop> (<exp>)`, $x+\cos$ `(<exp>)`, $x+\cos$ `(<preop> (<exp>))`, $x+\cos$ `(sin (<exp>))`, and finally at the 8th gene, the full decoded expression $x+\cos(\sin(x))$.

An important advantage of the grammatical evolution representation is that it uses a linear genome; as a result, GE can directly use all standard genetic algorithm operators. Furthermore, because of the simplicity of the linear representation, computer implementations of GE are relatively easy.

A curious fact about the previous example is that the decoding process terminated without translating all genes. This occurs because in GE the genetic operations do not know about the semantics of a genome until it is decoded, so the decoding process frequently ends up with a complete expression (final) without traversing the entire genome. These non-translated genes are common in nature, and such a segment is known as intron; nevertheless, in genetic algorithm the presence of intron seems undesirable because it might slow down the evolutionary process [1, 15]. In an attempt to overcome this problem, Ryan *et al.* introduced an extra genetic operator to prune the redundant genes with a certain probability. However, the prune operator just alleviates the problem; the introns will still appear during the evolutionary process.

A more serious problem with GE is the opposite situation of having redundant genes. During the decoding process, it may happen that a genome does not have sufficient genes to generate a complete expression (sentence), i.e., there are still non-terminals remaining. In this case, there are at least three choices: (1) eliminate that individual; (2) insert random genes at the end; (3) reuse its own genes by wrapping. The first and second option degrade the evolutionary process since either genetic material is lost or random genes are introduced (a kind of aggressive mutation). The third alternative seems to be the best one, but it might lead to an illegal expression—caused by unlimited growth—depending on the sequence of values in the genome; for example, using

204	143	56	223	15	7	76	191	233	1

Figure 2: A GE individual's genome.

the grammar described in Table 1, the genome `202, 145, 55` will always generate an infinite expression.

3.2 Grammatically-based GP

Being a formal tool to describe deep conceptual relationships, a grammatically-based representation is able to overcome the main flaws found in the canonical GP, CSS, and STGP representation. Even though GE employs a grammar—and thus can express deep constraints—, its linear genotype introduces some significant collateral issues.

This work adopts as representation scheme the proposal developed by Whigham [17, 18] to integrate a class of grammar, namely a context-free grammar, with genetic programming. But unlike GE, the tree structure is used to avoid the limitations of the linear encoding.

A formal description of a context-free grammar and its integration with GP are presented in the following.

3.2.1 Context-free Grammar — CFG

Formally, a grammar is a device for generating sentences—a string of words satisfying certain grammatical rules—e.g., a computer program or an arithmetic expression. The potentially infinite set of all such sentences defines the grammar's language. A CFG is described by a quadruple $G = (N, \Sigma, S, P)$, where:

- N is the (finite) alphabet of **non-terminal** symbols;
- Σ is the (finite) alphabet of **terminals**[3], such that $N \cap \Sigma = \emptyset$ and $N \cup \Sigma = V$, where V is the set of symbols of the language;
- S is the **start** symbol, $S \in N$;
- P is the set of **production rules** in the form $\alpha \rightarrow \beta$, where $\alpha \in N$ and $\beta \in V^*$. V^* denotes the set of all sentences composed of elements from V. Productions like $\{\alpha \rightarrow \beta, \alpha \rightarrow \gamma\}$ can also be written as $\{\alpha \rightarrow \beta|\gamma\}$.

The non-terminal symbols are those that express concepts, and eventually they will be replaced by terminal symbols in order to become functional. In English grammar the noun *verb* is a non-terminal that expresses a class of words (terminals) sharing the concept of verbs; in the same way, the non-terminal <*numerical*> in a CFG could denote a class relating to numbers, such as numerical constants, variables/attributes, and so forth. Moreover, a non-terminal can express another non-terminal; for example, the non-terminal <*numerical*> could have subclasses like <*integer*> or <*real*>.

On the other hand, the terminal symbols denote a concrete meaning; for instance, in a certain grammar the number π and the function sin (which returns a numerical value) could be terminal symbols pertaining to the non-terminal <*numerical*>.

The production rules define how non-terminals relate to terminals and other non-terminals. The first rule is actually given by the start symbol, which specifies how the sentence

[3]The meaning of the word *terminal* used here differs from its meaning in genetic programming. In the canonical GP, a *terminal* means a constant, variable, or a function that doesn't require arguments—i.e., a leaf node in the program tree. But in a grammar, a *terminal* is a more general term, and it can be not only a leaf node but also functions that require arguments (inner node).

must begin. The power of a CFG relies on its production rules, and such power is achieved when the production rules are combined together in order to form arbitrary sentences. The rules can be freely combined provided that a head of a rule (the non-terminal α) matches some non-terminal symbol in the body of another rule.

An example of a CFG grammar is shown in Figure 1. That grammar defines a language of unary and binary expressions for some arithmetic ($+$, $-$) and trigonometric (sin, cos) functions over the variable x.

Derivation Steps

A derivation step is the process of obtaining a sequence of symbols—also known as *word* or *string*—through the application of a production to a non-terminal symbol of the string undergoing the derivation.

The derivation begins through the application of a compatible production (head symbol matching the target non-terminal) to the start symbol S. If the resulting string has at least one non-terminal symbol, then the process of derivation continues applying production rules to replace each non-terminal symbol until a sentence is achieved[4], that is, a string without non-terminal symbols.

One interesting thing about the derivation is that a sequence of derivations can be expressed in a form of a tree. This makes the integration of a CFG with genetic programming very natural. In fact, a grammatically-based genome is a complete sequence of derivation steps represented by a tree—called syntactic or derivation tree.

3.2.2 *Integration with GP*

Grammar for the problem

The definition of a grammar is the first task in solving a problem. In a typical GP application, a CFG defines the functions, constants, variables, and also the relationships and constraints between them. A good grammar should have sufficient power (flexibility) to express the theoretical solution of the problem, while it should be parsimonious as much as possible to avoid unnecessary increase in the size of the search space.

A usual data classification problem, for instance, will normally include production rules to deal with conditionals (e.g. `if-then-else`), logicals (e.g. `or`, `and`, `not`), relationals (e.g. $<$, $>$, $=$), and sometimes arithmetic functions to process numerical data.

Creation of the Individuals

Each individual's genome contains a complete derivation tree (sentence), including the non-terminal symbols of the derivation[5]. In the initial population the trees are randomly created, i.e., during the derivation process any production rule—compatible with the current non-terminal—may be picked up at random.

Fitness Evaluation

The introduction of the CFG does not change the method of fitness evaluation of the canonical GP. In data classification, each individual is evaluated against a training set; the classifier is executed in every data training, and its fitness is proportional to the number of correct classifications (accuracy).

Although the non-terminal symbols are part of the program tree, they are not required at the execution time; they just ensure the structural integrity of the genome across the genetic operations.

Genetic Operators

The two main genetic operators in GP are *crossover* and *mutation*. The former recombines successful genetic material in order to hopefully create better individuals; the latter introduces new genetic material (increases the diversity) by introducing small random changes in some individuals. In their standard form, the crossover swaps random subtrees of two individuals (parents) while the mutation replaces a random subtree of an individual by an entirely new one.

To preserve the feasibility of the individuals it is necessary that each genetic operator respects the production rules of the grammar. In other words, all genetic operators must respect the constraints imposed by the non-terminals of the derivation tree. Therefore, for crossover and mutation the subtrees being swapped/replaced must have the same non-terminal root node (topmost node).

Figure 3 illustrates the crossover between the individual **A** and **B**. First, a random non-terminal node is selected in the tree **A**, namely <*class*>. Also, in the tree **B** a non-terminal having the same label is randomly selected. Finally, the two subtrees rooted by <*class*> are swapped, so creating two new individuals, **A'** and **B'**.

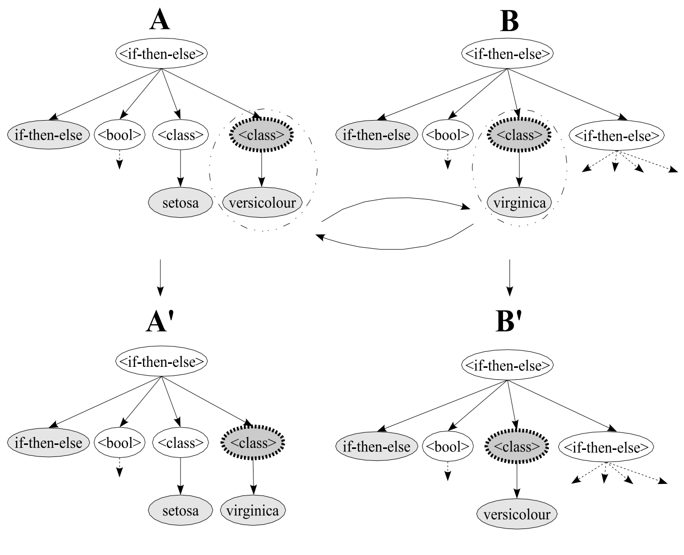

Figure 3: Grammatically-based Crossover

The mutation operator is exemplified in Figure 4. Again, a random non-terminal is selected in **A**, for example <*class*>. Then, the subtree rooted by that non-terminal is replaced by an entirely new subtree whose root node is also <*class*>. Note that the selected non-terminal was actually the start symbol used in the derivation process to create the new subtree.

[4]At each stage in the derivation the leftmost non-terminal is replaced.

[5]The non-terminal symbols are stored together in order to guarantee the genome feasibility under the application of the genetic operators.

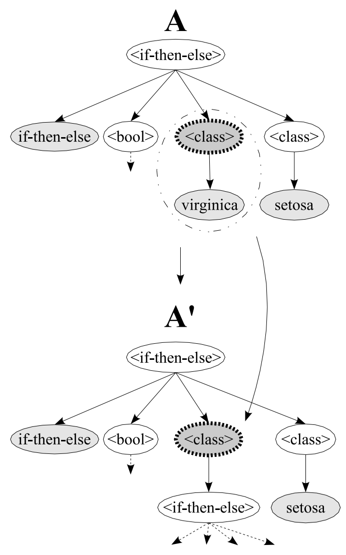

Figure 4: Grammatically-based Mutation

4. COEVOLUTION OF DATA SAMPLES AND CLASSIFIERS

Coevolution is the evolution acting upon two or more populations that interact closely with each another. It guides the evolutionary process towards a series of reciprocal changes in order to keep each population adapted with one another. There are different types of coevolution, ranging from mutual cooperation to competition; this section presents competitive coevolution in which a population of classifiers (classification trees) compete with a population of data samples (training data set).

4.1 Competitive Coevolution

To compete means that if an individual's fitness is increased then the competitor's fitness is decreased, and vice versa. In a data classification context, this means that if a classifier correctly classifies a data sample then its fitness is increased and the data sample's fitness is decreased; on the other hand, if a data sample induces the classifier to make a wrong classification then the data sample's fitness is increased but the classifier's fitness is decreased. According to Hillis [6], a competitive coevolutionary approach has two important properties: (1) it reduces the chance of getting stuck at local optima; and (2) increases the efficiency of the fitness evaluation because it makes the evolutionary process to focus on difficult data samples, i.e., those not easily classifiable.

Paredis [13], develops an interesting coevolutionary approach using genetic algorithm for data classification, in which a population of neural networks classifiers competitively coevolve with a population of input patterns; moreover, he proposed a limited-memory fitness evaluation scheme called *life-time fitness evaluation*.

The present paper brings Paredis's approach into a grammatically-based genetic programming context, where classification trees coevolve with training data samples. This coevolutionary process works as explained below.

In an iterative cycle, a pair of competitors are selected based on their fitness: one individual from the population of classifiers and the other one from the population of data samples. A classifier is declared the winner when it is able to correctly classify its opponent; in an analogous manner, a data sample defeats a classifier when it causes an incorrect classification. In both cases the winner's fitness is increased whereas the loser's fitness is decreased.

Note that only the population of classifiers undergoes the genetic operations since it does not make sense to change the data set. Although the data samples are not subject to the genetic changes, they impose contrary pressure on the population of classifiers on proportion that the most difficult samples—better fitness—are those that have more chance to compete, that is, those that are more frequently selected. But in the same way the best classifiers are those more often selected to compete. This arena of increasing bidirectional tension is the coevolutionary pressure; basically, the more an individual succeeds the more it must prove its skill.

The competition between classifiers and data samples is illustrated in Figure 5. The circles on the left represent the population of classifiers while the circles on the right represent the population of data samples; the best individuals of both populations are on the top. The lines symbolize the history of confrontation where the bolder ones indicate a greater number of encounters. An interesting fact is that the coevolutionary process smartly focuses on the harder data samples rather than wasting unnecessary resources on the easier ones. As a result, the overall evaluation process is more efficient than the traditional method where each classifier is always evaluated against every data sample.

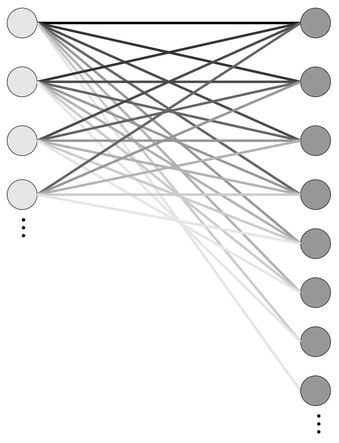

Figure 5: Illustration of competitions between classifiers and data samples.

4.2 Life-Time Fitness Evaluation — LTFE

Instead of assigning a fixed number to indicate the fitness of an individual after its evaluation, Paredis proposed an accumulative scheme inspired by the concept of "energy", in which the adaptive degree of an individual –and so its reproduction success– is measured by the quantity of energy accumulated during its existence. This scheme is named life-time fitness evaluation (LTFE) [12].

In genetic programming the LTFE works as follows. A finite and discrete number determines the "memory" capacity, i.e., the number of competition results that will be stored for an individual as its fitness. The most recent results replace the oldest ones in order to keep the size of the history constant. For each confront it is stored **win** (1) if an individual has defeated the opponent or **not win** (0) otherwise. Thus, the difference between the number of victories and the number of defeats defines an individual's fitness.

An important property of the LTFE is its ability to deal with noise. In the coevolutionary scheme, a noisy data sample could easily degrade the evolutionary process since much effort would be wasted in order to dominate such a "difficult" sample[6]. However, a finite memory, which forgets the older encounters, reduces the distance between the difficult individuals and the easy ones, thus reducing the –possibly excessive– degree of focusing on the hardest data samples.

4.3 Coevolution of Data Samples and Classifiers

The kernel of Paredis's coevolutionary process adjusted for data classification is given in Algorithm 1, which is repeated until a termination criterion—such as "solution found" or "time limit"—is reached. It is assumed that both populations, classifiers and data samples, are already initialized and their individuals evaluated. The first evaluation of a recently created classifier is done simply by putting it to compete against a specified number of random data samples[7]; at the same time, the fitness of each data sample is also updated. Whenever a competition happens, its result is inserted into the fitness history of both individuals, and the oldest record is discarded in order to keep constant the history lenght –first in first out (*FIFO*).

Algorithm 1 Coevolutionary cycle with LTFE.

```
while cycle is incomplete do
  Competitor $A \leftarrow$ select a promising classifier.
  Competitor $B \leftarrow$ select a promising data sample.
  Promote the competition between $A$ and $B$.
  if $A$ classifies $B$ then
    Insert victory into $A$'s fitness history.
    Insert defeat into $B$'s fitness history.
  else
    Insert defeat into $A$'s fitness history.
    Insert victory into $B$'s fitness history.
  end if
end while
Select two classification trees, $X_1$ and $X_2$, for mating.
Recombine $X_1$ with $X_2$ to create the children $X_1'$ and $X_2'$.
Apply other operators (e.g. mutation) on $X_1'$ and $X_2'$.
Evaluate $X_1'$ and $X_2'$, and insert them into the population.
```

Formation of the Populations

While the initial population of classifiers consists of random derivation trees of a certain language defined by a context-free grammar, the population of data samples is composed by the (immutable) training samples of the problem's data set.

[6]Since a noisy data sample is usually far away from the nature/essence of the data set, it is certainly very difficult to classify.

[7]The number of data samples, which defines the "length" of the *cycle*, is a parameter that may vary depending on the problem. Paredis suggests 20 competitions per cycle.

Fitness Calculation

Being a set of historical results rather than a single value, the fitness must be transformed to be used by the selection process; basically, the number of wins is summed and then normalized. For a data sample individual, its resulting fitness f_{sample} is defined as follows:

$$f_{sample} = \frac{1}{n_{cycle}} \sum_{i=1}^{n_{cycle}} H_i$$

where n_{cycle} is the size of the *cycle* (normally 20) and H_i is the i-th value of the history of encounters—0 if it was correctly classified; 1 otherwise.

The resulting fitness of a classifier is obtained in an analogous manner, but it also includes a measure of complexity, which is expressed in terms of *number of nodes* and *attributes used*, in order to penalize complex trees; therefore, the f_{classf} is given by:

$$f_{classf} = \frac{(1-\alpha)}{n_{cycle}} \sum_{i=1}^{n_{cycle}} H_i - \frac{\alpha}{2}\left(\frac{nodes}{nodes_{max}} + \frac{attributes}{attributes_{max}}\right)$$

where $node$ and $node_{max}$ are respectively the number of nodes of the tree and the maximum number of nodes so far; similarly, $attributes$ and $attributes_{max}$ are the number of attributes of the tree and the maximum number of attributes available in the data set. The α parameter controls the level of penalization.

Selection Scheme

In the competition stage a *rank-based* selection is used for (both populations); however, a *tournament* selection is used in the recombination stage for the population of classifiers).

Reproduction Method

As in Paredis's work, the *steady-state* reproduction is adopted here, so each time an individual is created it is immediately inserted into the current population. In order to keep the size of the population constant, the worst individuals are removed.

5. EXPERIMENTS

This section presents some data classification experiments (from the machine learning repository of the University of California [2]) in order to observe the performance of the proposed technique.

5.1 Methodology

When not fixed by the problem, the original data set was divided into two subsets: the training set, with 2/3 of the records, and the test set, with 1/3 of the records. Thirty independent runs were performed for each experiment; this means different seeds and random data distribution for the training and test sets for each execution. Each table of results summarizes the averages of the accuracy on the training and test sets, tree size, and evaluations[8].

Some parameters remained constant throughout the experiments, namely, *probability of crossover* (90%), *probability of mutation* (5%), *size of tournament* (2 individuals), and *size of the Paredis's cycle* (20).

[8]The number of evaluations is taken at the generation where the best individual is found.

The algorithm was implemented in C++ language on an AMD 2.2GHz machine with 2GB RAM running GNU/Linux.

Grammar

All problems were carried out upon a rather generic context-free grammar—maybe even redundant regarding the current experiments. The language defined by such grammar includes the conditional *if-then-else*, relationals ($=$, $\neq$, $<$, $\leq$, $>$, and $\geq$), logicals (*or*, *and*, and *not*), arithmetics ($+$, $-$, $\times$, and $\div$), and ephemeral constants. Also, this grammar allowed for inter-attribute comparison and arithmetic operations upon them.

5.2 Congressional Voting Records Data Set

This data set includes votes for each of the U.S. House of Representatives Congressmen on the 16 key votes identified by the *Congressional Quarterly Almanac*. There are three (there were originally nine) different types of votes: *yes*, *no*, and *unknown*. The goal of this problem is, based on the votes on those 16 points (attributes), to identify which political party (democrat or republican) would make such votes. The data set has a total of 435 records.

The first experiment was performed with the following parameters: population of 300 individuals, maximum of 1000 generations, and $\alpha = 0.15$ (coefficient of complexity). Table 1 summarizes the result.

	Accuracy	Tree Size	Evals ($\times 10^6$)
Train. Set	0.984 ± 0.013	74.3 ± 41.0	5.22 ± 3.17
Test Set	0.975 ± 0.019		

Average execution time: 11 min.

Table 1: Congressional Voting Records I

The proposed approach was able to produce very accurate results; for comparison, C4.5 [14], also using 1/3 records for the test set, achieved 94.5% on the test set (98.3% on the training set). However, C4.5 produced trees with average size of 25.9 nodes. So, in order to reduce the size of the solutions, the coefficient of penalization α was increased to 0.2. The result can be visualized in Table 2, in which smaller trees (mean = 26.4) were created without significant loss in accuracy.

	Accuracy	Tree Size	Evals ($\times 10^6$)
Train. Set	0.975 ± 0.011	26.4 ± 6.1	3.91 ± 2.07
Test Set	0.971 ± 0.013		

Average execution time: 5 min.

Table 2: Congressional Voting Records II

5.3 Monk's Problems Data Set

The Monk's problem is a collection of three artificially generated data sets created by Thrun *et al.* [16] in order to evaluate the performance of different learning algorithms. The data sets contain 432 records, 7 attributes, and 2 classes. Each problem has a specific purpose (level of difficulty) and each one defines its own training set. The test set is always the whole 432 samples.

Monk's Problem I

Monks I is generated in standard disjunctive normal form and the problem is supposed to be easily accomplished by most of the learning algorithms. The training set consists of 124 records.

The settings for this experiment were: population of 500 individuals, maximum of 100 generations, and $\alpha = 0.2$. Table 3 shows that in fact the problem is somewhat easy; all runs found the perfect solution. However, as reported in [16], only 9 out of 25 learning algorithms could reach 100% of accuracy.

	Accuracy	Tree Size	Evals ($\times 10^5$)
Train. Set	1.000 ± 0.000	14.0 ± 5.6	1.42 ± 1.02
Test Set	1.000 ± 0.000		

Average execution time: 13 sec.

Table 3: Monk's problem I

Monk's Problem II

This second data set, on the other hand, is similar to parity problems, and it combines the attributes in a certain way that makes it hard to describe relying on the given attributes only [9, 8]. The training set has 169 records.

The parameters of this experiment were chosen regarding the complicated nature of this problem: population of 500 individuals, maximum of 5000 generations, and a relaxed $\alpha = 0.05$. The result is presented in Table 4.

	Accuracy	Tree Size	Evals ($\times 10^7$)
Train. Set	0.999 ± 0.002	195.1 ± 214.3	1.90 ± 0.52
Test Set	0.970 ± 0.019		

Average execution time: 42 min.

Table 4: Monk's problem II

According to [16], only 4 (out of 25) algorithms were able to achieve 100% of classification accuracy; moreover, only one of them produces symbolic solutions[9]. The best accuracy of the other 21 algorithms was 93.1%.

Monk's Problem III

Finally, like *Monks I*, Monk's problem III is also created in standard disjunctive normal form, however, its training set has 5% of misclassifications; moreover, there are only 122 training data samples. The purpose of this data set is to evaluate the capacity of generalization of an algorithm.

This experiment used a population of 500 individuals, maximum 500 generations, and $\alpha = 0.15$. Table 5 summarizes the results, and one can see that the accuracy in the training set is lower than that of the test set. This is due to the fact that the training set contains misclassified data, and a good generalizer tends to ignore noisy data.

	Accuracy	Tree Size	Evals ($\times 10^6$)
Train. Set	0.961 ± 0.032	37.1 ± 29.8	3.40 ± 3.25
Test Set	0.995 ± 0.015		

Average execution time: 5 min.

Table 5: Monk's problem III

Although the proposed algorithm could not produce a perfect score, it has performed very well, showing a great capacity of generalization. In the report provided by Thrun *et*

[9]It was the AQ17-DCI algorithm, which can directly deal with attributes, and so derives new ones when necessary.

al. [16], only 5 algorithms were able to reach 100% of accuracy. The best result for the other 20 algorithms was 97.2%.

6. CONCLUSIONS

This work was built upon three pillars, namely, genetic programming, grammatically-based representation, and competitive coevolution. These techniques were combined together in order to perform the data classification task.

Genetic programming provided a robust framework in which a population of symbolic classification trees evolves through Darwinian principle of natural selection. A formal context-free grammar structure replaced the canonical GP representation; as a result, the classifiers were able to be described in an arbitrary language, possibly consisting of deep conceptual relationships. Finally, a coevolutionary scheme promoting competition between classifiers and data samples was introduced in order to improve the process by focusing on the difficult data samples.

Some experiments were made and the results, when compared with other algorithms found in the literature, were very encouraging. The first experiment showed that, though C4.5 is faster, the proposed algorithm produced better classification accuracy. The second experiment, which is a collection of three problems, revealed the ability of the proposed algorithm to deal with a known difficult problem (parity problem) and with the presence of noises (generalization).

Future work includes: (1) studying the possibility of evolving the production rules of the grammar simultaneously with the standard evolutionary process, so that the rules become more optimized to the current problem; (2) evaluating the proposed algorithm on other data sets in order to find out its strengths and also its weaknesses; (3) investigating the coevolutionary process to detect—and to try to solve—whether there are occurrences of cycles of alternating strategies, that is, pattern of solutions that are rediscovered repeatedly and do not produce continuous increase of complexity (classification accuracy) during the process.

Acknowledgments

The authors thank the reviewers for the corrections and suggestions, and the Brazilian agencies *National Petroleum Agency* (ANP) and CNPq (grant 311651/2006-2).

7. REFERENCES

[1] David Andre and Astro Teller. A study in program response and the negative effects of introns in genetic programming. In *Genetic Programming 1996: Proc. of the First Annual Conference*, pages 12–20, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

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

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, Reading, Mass., 1989.

[4] Frédéric Gruau. On using syntactic constraints with genetic programming. In *Advances in genetic programming: volume 2*, pages 377–394, Cambridge, MA, USA, 1996. MIT Press.

[5] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 271–278, Pittsburgh, PA, USA, 15-19 1995. Morgan Kaufmann.

[6] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Phys. D*, 42(1-3):228–234, 1990.

[7] John H. Holland. *Adpatation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[8] Lorenz Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, pages 158–166, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998.

[9] John R. Koza. *Genetic programming: On the programming of computers by natural selection.* MIT Press, Cambridge, Mass., 1992.

[10] David J. Montana. Strongly typed genetic programming. Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 1994.

[11] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[12] Jan Paredis. The evolution of behavior: some experiments. In *Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, pages 419–426. MIT Press, 1991.

[13] Jan Paredis. Steps towards co-evolutionary classification neural networks. In *Proc. of the Fourth Intl. Workshop on the Synthesis and Simulation of Living Systems*, pages 102–108, 1994.

[14] J. Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[15] Conor Ryan, J. J. Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, 14-15 1998. Springer-Verlag.

[16] S. B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Džeroski, S. E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R. S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek, and J. Zhang. The MONK's problems: A performance comparison of different learning algorithms. Technical Report CS-91-197, Pittsburgh, PA, 1991.

[17] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 1995.

[18] Peter Alexander Whigham. *Grammatical Bias for Evolutionary Learning.* PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia, 14 October 1996.