RESEARCH PAPERS OF WROCŁAW UNIVERSITY OF ECONOMICS No. 104

Business Informatics 16 2010

Arnaud Quirin[*], Jerzy Korczak[**]

[*]European Centre for Soft Computing, Mieres, Spain
arnaud.quirin@softcomputing.es
[**]Wrocław University of Economics, Wrocław, Poland
jerzy.korczak@ue.wroc.pl

GRAMGEN: A GENETIC PROGRAMMING SYSTEM BASED ON CONTEXT FREE GRAMMAR

Abstract: In this paper, a new genetic programming system, called GramGen, is described. The system combines context free grammar (CFG) with genetic programming and uses an extended operator set. The objective of the grammar is to limit the size of the search space by allowing the user to define constraints related to the structure or the simplicity of the discovered formulas. These constraints are taken into account by the use of specific genetic operators. The experiments show that the proposed system allows users not only to discover new knowledge but also improve the performance of existing ones.

Keywords: genetic algorithms, genetic operators, context free grammar, knowledge discovery, evolution-based software.

1. Introduction

A genetic programming algorithm (GP) is a kind of evolutionary algorithm in which the genetic individuals correspond to functions or programs [Koza 1992]. GP offers several interesting advantages in terms of knowledge representation and knowledge discovery. The rule representation by syntactic trees is easy to understand and facilitates knowledge validation by a human expert. On the other hand, an evolutionary algorithm provides an increased resistance of discovery with regards to noisy data and local minima. The ability to customize genetic operators allows a deeper interaction with the human expert or the domain knowledge (such as specific convergence criteria, fitness function able to deal with understandability measures for the trees such as size, depth, balancing).

Context free grammar-based [Freeman 1998; Javed et al. 2004] GP allows the user to directly influence and constrain the evolutionary search to produce the programs he really needs for his domain-specific application. Up to our knowledge, this thematic is not well extended in the literature. For instance, García-Arnau et al. [2007] developed such an approach which also integrates a parameter to control the maximum size of the generated trees. However, they considered only the size

measured as the height of the tree. In our approach, we consider both the total number of terminals in a tree and the height of the trees as two different criteria for the user. Attention has been given to the crossover operator [Manrique et al. 2005] to guarantee the consistency of this genetic operation, but less work has been dedicated to the mutation operator. In our approach, we propose a consistent crossover operator as well as a highly customizable and consistent mutation operator.

In this paper, a grammar-based GP system, called **GramGen**, is proposed to limit the size of the search space and allow defining constraints related to the structure and complexity of the formulas. In the system, the constraints are taken into account by applying specific genetic operators to simplify generated formulas, and in consequence, make them more comprehensive.

To introduce the problem of knowledge representation in GP systems, let us define a few basic terms, notably:

- **Genotypic tree.** A tree corresponding to the content of the genetic chromosomes. This tree is related to the grammar and is used to facilitate the generation of new individuals.
- **Phenotypic tree.** A tree corresponding to the interpretation of the genotypic tree. This tree is derived from the genotypic tree and encodes the function given to the user.
- **Non terminal symbol**. The non terminal symbols are nodes of the genotypic trees and they are only useful during the derivation step, in which the production rules of the grammar are applied.
- **Terminal symbol** or **terminal operator**. The terminal symbols are nodes existing both in a genotypic tree and in a phenotypic tree. In this paper, this term does not indicate the value of a node, but the corresponding keyword. For instance, *opMUL* refers to the multiplication, *opSIN* to the sinus and *opCST* to an instantiated constant.
- **Node operator** or **functions**. Node operators correspond to functions with an arity greater than 0, such as *opMUL*, *opSIN*, etc. They are only used in the leafs in the genotypic trees (and not in the phenotypic trees).
- **Leaf terminal**. Leaf terminals are leafs in the phenotypic tree and correspond to functions of arity 0, such as *opCST* (instantiated constants) or *opARG* (instantiated arguments).
- **Grammatical disjunction**. Right part of a production rule containing one or several conjunctions. This part replaces the left part during a derivation.
- **Grammatical conjunction**. Part of a disjunction corresponding to a terminal or a non terminal symbol.

In the next section the **GramGen** system is presented. In section 3, the formalism of the grammar is described. Section 4 introduces the terminal operators used in the algorithm, the *opPUSH* operator and the *opPOP* operator followed by the initialisation, the crossover and the mutation operators are described. Finally, in the last section, a simple example is discussed illustrating the ability of our algorithm to produce comprehensible rules.

2. Main algorithm

The general principle of rule discovery by GP follows the principle of evolution-based algorithms [Goldberg 1989], except that here, the rules introduced by the algorithm correspond to tree structures including node operators and leaf terminals.

GramGen uses this principle, and in addition, implements a set of constraints on the tree representation using a grammar initially defined by the user. For instance, the following example shows a small grammar which can be used to learn equation having only addition and substraction.

$S \rightarrow E$ # Start symbol
$E \rightarrow O2\ E\ E \mid opARG$
$O2 \rightarrow opADD \mid opSUB$

Another point is that the individuals of the population are ordered by their fitness values as an efficient way to find similar individuals. This measure is simply used as an indicator of a premature convergence of the population and is shown to the user.

Algorithm A1 describes the main algorithm of **GramGen**.

GramGen applies a grammar, parameterized by the user before starting the algorithm to discover the rules. The genetic operators are strongly based on this grammar. Thus, during the initialisation step, the crossover step, or the mutation step, the constraints defined by the grammar are always true at the output of the operators. These constraints are defined in terms of grammatical constraints (the numerator of a ratio must not have multiplicative signs) and probability constraints (the multiplicative and addition signs must have the same probability of appearance). In particular, we have proposed two ways to guarantee that the genetic operators produce only correct offsprings. The *passive method* checks each produced individual and deletes the trees which do not satisfy the grammatical constraints. The *active method* ensures that the operators produce directly correct individuals in terms of grammatical constraints and predefined probabilities. These last operators are more complex to describe, but as they do not lose time by generating and deleting useless individuals, thus they have been selected in the final implementation of **GramGen**.

Function GramGen(*Prop*$_{\chi}$, *Prop*$_{\rho}$, *Prop*$_{\mu}$**)**
Prop$_{\chi}$ *is the proportion of crossovers in the population*
Prop$_{\rho}$ *is the proportion of reproductions in the population*
Prop$_{\mu}$ *is the proportion of mutations in the population*
Result *- I, the function (tree) solving the given problem*

Initialize the algorithm (*gen*=0, *pop*={})
Randomly create an initial population *pop*
Evaluate the *fitness* of each individual in the population

repeat
  Begin a generation (*indiv*=0, pop_{new}={})
  while indiv/size(pop) < $Prop_{\chi}$
    Choose two individuals from *pop* based on the crossover selection strategy
    Carry out the crossover (see the section 4)
    Insert the two offsprings into pop_{new} (*indiv*=*indiv*+2)
  end while
  while indiv/size(pop) < $Prop_{\rho}$
    Choose an individual from *pop* based on the reproduction selection strategy
    Copy this individual into pop_{new} (*indiv*=*indiv*+1)
    end while
    while indiv/size(pop) < $Prop_{\mu}$
    Choose an individual from pop_{new} based on the mutation selection strategy
    Choose randomly a type of mutation among the three available (see the section 4)
    Carry out the mutation
    Insert the new individual into pop_{new} (*indiv*=*indiv*+1)
  end while
  Compute the new population (replacement) : *pop*=**Recycle**(*pop*, pop_{new})
  Evaluate the *fitness* of each individual in the population
  Order the individuals of the population by their *fitness*
  Compute the values of the termination criteria (number of generations, best *fitness* exceeding a threshold, average *fitness* exceeding a threshold)
  gen=gen+1
until One of the termination criteria are satisfied
Return the best individual *I* according to its *fitness*

Algorithm A1. Algorithm **GramGen**

Let us describe the key concepts of the algorithm. The fitness function of **GramGen** measures the relative quality of an individual by comparing it to other individuals in the population. The fitness function is the ratio of the wealth of an individual and the sum of the wealth of all individuals of the whole population. In our case, this evaluation is computed for each individual using the following formula:

$$E(i) = C_{\text{fun}}E_{\text{fun}}(i) + C_{\text{size}}E_{\text{size}}(i) + C_{\text{cst}}E_{\text{cst}}(i) + C_{\text{arg}}E_{\text{arg}}(i), \quad (1)$$

where $E_{\text{fun}}(i)$, $E_{\text{size}}(i)$, $E_{\text{cst}}(i)$ and $E_{\text{arg}}(i)$ are the evaluation of the individual i according to specific constraints, and C_{fun}, C_{size}, C_{cst} and C_{arg} are the weights set by the user according to his appreciation for each constraint. The values of each specific evaluation function are between 0 and 1.

$E_{\text{fun}}(i)$ computes the accuracy between the expected value of a training sample and the value obtained using the formula encoded in the individual i. The average of the sum of the error squared obtained on the training set is used as the result of $E_{\text{fun}}(i)$. It should be noted that in regression problems, the error corresponds to $e^{-|D|}$, where

D is the difference between the obtained and the expected value. In classification problems, for an expected class, the error is 0 for a positive value and 1 for a negative value. And, for an unexpected class, the error is equal to 1 for a positive value and 0 for a negative value.

$E_{size}(i)$ computes the score of an individual according to an expected size, which corresponds to the number of nodes in the tree. This enables the user to specify a constraint on the size of the tree, which influences to the understandability of the obtained formula. In general, the shorter a formula is, the more understandable it will be, but the accuracy will also decrease. The user inputs an ideal number of nodes, X, and a maximal number of nodes, M. Let a given formula be i with s nodes. If s is between 0 and X, the score is computed proportionally between 0 and 1. If s is between X and M, the score is computed proportionally between 1 and 0. And, if s is above M, the score is set to 0.

$E_{cst}(i)$ and $E_{arg}(i)$ compute the score of an individual according to the number of constants and to the number of arguments used in the formulas. When a formula contains more constants, the accuracy is higher, but it will be more difficult and specific to the training set. Comparatively, when a formula uses more arguments (attributes of a training sample), this formula will be more complex and more difficult to understand. The user can input an upper number of constants and/or arguments, and the score is computed as before.

Consequently, the evaluation function used in **GramGen** is a trade-off between the accuracy and the constraints defined by the user in terms of understandability and readability of the obtained formulas.

3. Formalism of the grammar

Regression systems based on GP usually amplify the size of the trees during the search of a reliable solution. For instance, a research work presented by Ross [Ross et al. 2002] shows a tree requiring about fifty nodes to be effective in a classification problem for a real-world application. The produced tree does not deliver a clear and intuitive explanation to users. Type-based genetic programming is often difficult to understand, especially for users without extensive GP experience who need to design grammars. However, in these kinds of problems, a rigorous interpretation of the generated functions by a human expert is required for the validation of these functions. To simplify the trees, some constraints have often been proposed, as for instance those described by Montana [1994]. In his project, the nodes are associated with data types and only the authorized grammatical constructions are admitted. However, in many regression or classification problems, the data have often the same format and this kind of type assignment is not required.

In our approach, the well-known Context Free Grammar (CFG) has been applied [Freeman 1998; Javed et al. 2004]. This grammar is simple, general and can be put in the normal form to be fast and effective for the parsing operators.

Formally, a CFG can be defined as a quadruplet $G = (V\_t, V\_n, P, S)$, where:

- V_t is a finite set of terminals,
- V_n is a finite set of non-terminals,
- P is a finite set of production rules,
- S is an element of V_n and corresponds to the *start symbol*.

The elements of P are represented by $V_n \rightarrow (V_t \cup V_n)^*$. Below, an example of a CFG grammar is given:

- $V_t = \{opADD, opARG, opCST\}$,
- $V_n = \{S\}$,
- $P = \{S \rightarrow opADD\ S\ S, S \rightarrow opCST, S \rightarrow opARG\}$,
- $S = S$.

Figure 1 illustrates this CFG. The operator *opADD* corresponds to the addition between two numbers and the operator *opARG* corresponds to an argument of the function (the real index of the argument is instantiated in the phenotypic tree). The operator *opCST* corresponds to a constant (the real value of the constant is instantiated in the phenotypic tree). Grammars like this are able to derive trees of variable sizes.

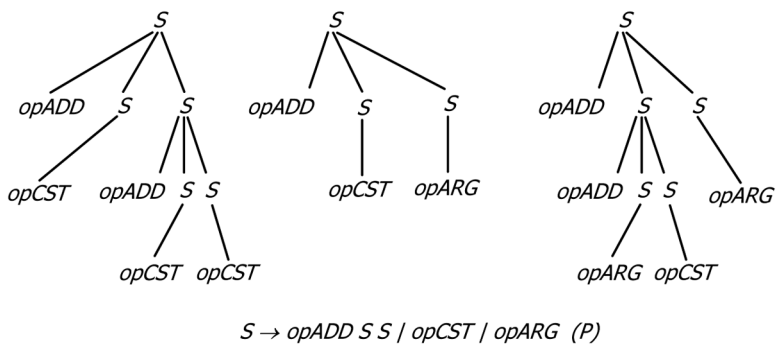

Figure 1. Several genotypic trees produced using a derivation rule and an instance of a production rule

In **GramGen**, the start symbol is derived using the production rules until the obtained set contains only terminal symbols. The trace of this derivation can be reproduced in the form of a tree, called in the paper a *derivation tree* or a *genotypic tree*. Thus, the trace of the rule $A \rightarrow B$ is a tree with a root node A connected to a child node B. A rule $A \rightarrow B\ C$ is a tree with a root node A connected to two child nodes B and C. At a given time, a non terminal symbol has to be derived, in one or more symbols, using only one of the production rules. The symbol "|" is used to separate several production rules (disjunctions). The creation of the genetic individuals is carried out in two steps (Figure 2): the grammar is firstly used to define a genotypic tree A_G, and then this tree is converted into a phenotypic tree A_P corresponding to the function that the algorithm is looking for. The tree A_P is obtained by replacing any genotypic subtree A'_G containing a root node X and N +1 edges by a phenotypic subtree A'_P where the root corresponds to the first edge of A'_G (corresponding to

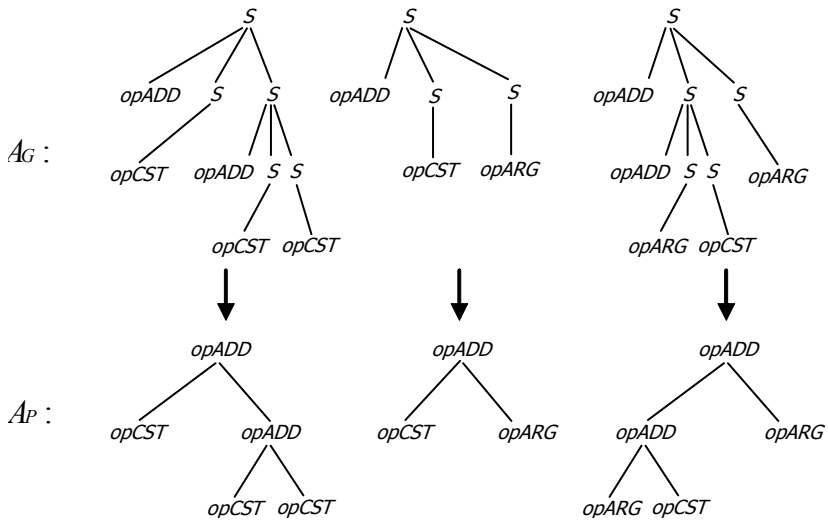

Figure 2. Conversion of genotypic trees into phenotypic trees

a function of arity *N*), and where the *N* edges correspond to the number of edges 2 to $N+1$ of the tree A'_G. The complete phenotypic tree is obtained by performing all the replacements but the grammar is not required during this process. The Polish notation is applied: the node pointed by the first edge of each node encodes the function and the following nodes its arguments.

This grammar needs consistent genetic operators to work with. By *consistent* we mean that the application of the crossover or the mutation genetic operators keeps the generated trees compatible with the defined grammar. The next two sections present the standard and the special terminal operators composing the grammar, while the subsequent three sections present the initialisation, the crossover and the mutation genetic operators.

4. The GramGen terminal and genetic operators

4.1. Terminal operators

The terminal operators correspond to atomic units for the phenotypic trees produced by GP. Two kinds of terminals can be distinguished: the leaf terminals (arguments or constants), and the functions. Many operators have been implemented, and they are mainly mathematical, boolean and statistical. In the experiments some mathematical operators, some operators of arity 3 and some operators of arity *N* have been used. For instance, in addition to the implementation of common functions, a few useful functions for regression problems have been proposed such as the functions of arity *N* with an unspecified number of parameters in the input, in particular *opSOMM* (sum of the arguments) or *opAVG* (average of the arguments).

The user may select the terminals to use in the genetic programs, either implicitly in the form of probabilities in the generated trees, or directly by the definition of the production rules. The syntax of the grammar has an implicit way to define the preferences of the apparition of a given terminal in the generated trees. If the user decides to repeat a given symbol (a terminal or a non terminal symbol) during the writing of a production rules, this symbol has a higher probability to be selected. For instance, if the user writes the rule $S \rightarrow opADD\ opADD\ opADD\ opSUB$, the symbol *opADD* (the addition operator) is selected in 75% of the cases, and the symbol *opSUB* (the substraction operator) is selected in the remaining 25% of the cases.

If needed, the setting up of the appearance probability of a terminal symbol (or a non terminal symbol) can be defined by repeating the occurrence of this symbol several times in the production rule.

4.2. Queue operators

In some cases, the user could need to manipulate a structure having the behaviour of a list, a file, etc. For instance, in some problems requiring to sort values in a table, or to manipulate a specific kind of data (spectral signal, ...), operators to facilitate the management of *First In, First Out* (FIFO) queues are needed. FIFO queues allow a function to build tables where its size is known only during their execution. So their size can dynamically increase or decrease depending on the input attributes of the symbolic expression. Two queue operators are designed: the *opPUSH* and *opPOP* operators are these special operators. As a tree is interpreted by depth-first search (the child nodes of a given node are computed before its parent), it is consequently possible for a node to carry out computations on a file encoded in one of its child nodes. Figure 3 shows a grammar construct using such operators, a genotypic tree built using this grammar, the related phenotypic tree and its semantic interpretation.

If needed, some inter-type conversions may occur in the nodes during the assignment of a value. For instance, a strictly positive constant returned by a mathematical operator is converted into a boolean constant equal to *true*. A null or negative value is converted into *false*, and the values *true* and *false* are respectively converted into 1 and 0. Lastly, a parameter is integrated into each one of these operators to define if the operator is commutative or not. This parameter is applied during the structural comparison of two trees, and that is used as a diversity criterion in the termination operator.

4.3. Initialisation operator

In **GramGen**, the initialisation operator for genetic individuals has to select the grammatical rules as well as the order of their application regarding a given number of constraints. This operator is applied to create the first genetic population or new edges claimed by the mutation operator. If several possibilities arise, the choice of

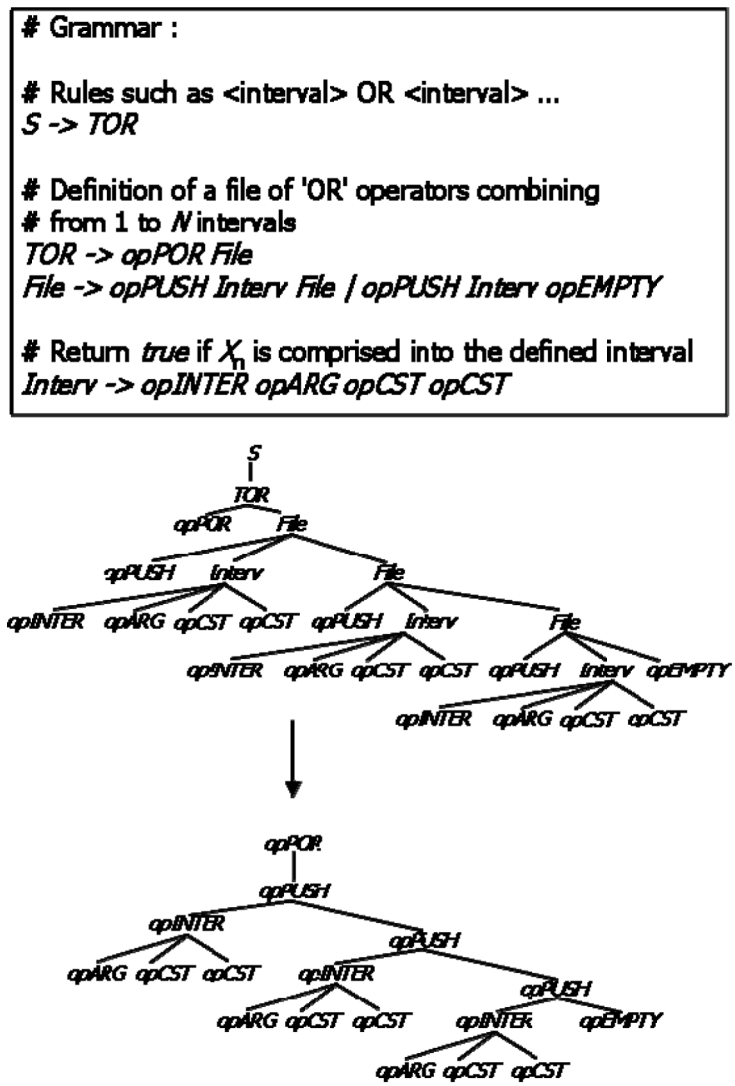

Figure 3. Grammar using the *opPUSH* operator

***Sub-Function Height*(*X*)** – *Function computing the minimal height of a symbol X*
H(*X*) *is an attribute of the symbol X*
Result – *The height of the symbol X*

let $h := \infty$
for each disjunction *D* of the rule *X*
  let *hc* := 0
  for each conjunction *C* of *D*
    hc := **Max**(*hc*, *H*(*C*))
  end for
  h := **Min**(*h*, *hc*)
end for
Height(*X*) := *h* +1

***Sub-Function Size*(*X*)** – *Function computing the minimal size of a tree generated by the derivation of a symbol X*
T(X) is an attribute of the symbol X
Result – *The size of the symbol X*

let $s = \infty$
for each disjunction *D* of the rule *X*
  let *sc* = 0
  for each conjunction *C* of *D*
    sc = *sc* + *T*(*C*)
  end for
  s = **Min**(*s*, *sc*)
end for
Size(*X*) := *s*

***Function InitSymbols*(*L*)** – *Function used to initialize the symbols of the grammar*
L is the list of the symbols $V_t \cup V_n$ *of a grammar G*
Result *–- A parameterized grammar: the height and the minimal size of each symbols of L*
for each symbol *X* of *L*
  if *X* is terminal
    H(*X*) = 0
    T(*X*) = 1
  else
    $H(X) = \infty$
    $T(X) = \infty$
  end if
end for
repeat
  for each non terminal symbol *X* of *L*
    H(*X*) := **Height(*X*)**
    T(*X*) := **Size(*X*)**
  end for
until the attributes of the symbols in *L* have converged
Returns the parameterized grammar

Algorithm A2. Function InitSymbols

the production rule is done according to several criteria, based on the size of the trees and the desired depth. The two principal steps devoted to the construction of the new individuals are as follows:

- determination of the height $H(X)$ or the smallest number of terminals $T(X)$ that it is possible to generate in the best case for each symbol X,
- during the construction of a subtree or a complete tree, determination of the symbol to use according to a given random probability related to $H(X)$ or $T(X)$.

The first step is performed only once, at the time of the grammar parameterization. This step is described in Algorithm A2. Figure 4 shows a parameterized grammar. The parameterization concerns the height and the minimal number of symbols that can be obtained in a genotypic tree derived from a given symbol. The computing of the maximum values is not very interesting for most grammars because they are infinite. The main interest of this algorithm is that it converges even in the case of *full-recursive* grammar rules (for instance, $A \rightarrow A$). In this case, these rules are automatically ignored.

$$
\begin{array}{l}
S \rightarrow E \\
E \rightarrow OEE \mid V \\
O \rightarrow opADD \mid opMUL \\
V \rightarrow opARG \mid opCST \mid 5.34
\end{array}
$$

Symbol	H(Symbol)	T(Symbol)
S	3	1
E	2	1
O	1	1
V	1	1
opADD	0	1
opMUL	0	1
opARG	0	1
opCST	0	1
5.34	0	1

Figure 4. Example of a grammar and its parameterization

The second step is performed during the creation of the individuals or each time a genetic operator requires the creation of a subtree. This step is presented in algorithm A3. The parameters of the algorithm are the following: a grammar, a non-terminal symbol (either the start symbol *S*, or another) and the constraints defining tree size and tree height. The result of the algorithm is a complete genotypic tree or a subtree which can then be included in a larger tree, which will be converted into a phenotypic tree before the evaluation of the individual. The algorithm performs its computing in an exact constraint environment, that is, the size specified by the user in terms of number of nodes is always respected. For instance, if the user selects an even size N, and that grammar can only produce trees with odd sizes, then there is a probability of 0.5 that an individual of size $N - 1$ is produced and 0.5 for an individual of size $N + 1$. If the user specifies a null or a negative value for the size, the produced tree will be as small as possible. If some variable sizes are required, the user has to choose a range of acceptable values, for instance, trees containing from 3 to 15 nodes. Then

the algorithm randomly selects a value in this range and uses it as parameter. The algorithm is called as many times as required to constitute a complete population. Thus, it is possible to obtain a population including trees where sizes can be specified by various probability functions: uniform, linear, Gaussian, etc. In our case, a linear function is used.

The determination of the non terminal symbol to derive, when the current derivation tree contains several symbols (known the derivation style), is not performed from the left (*leftmost derivation*), nor from the right (*rightmost derivation*). In fact, the best results have been obtained each time by randomly choosing the non terminal symbol in the tree currently in derivation. Moreover, that guarantees some diversity in the pool, even if the grammar is badly written.

***Function TreeCreation*(*G*, *A*, f_{ask})**
Parameters – A parameterized grammar G, a non terminal symbol A, the requested size or height f_{ask} for the generated tree
f(*X*) *is the criterion to optimize. Either H*(*X*) *(the height of the subtree generated by the symbol X), or T*(*X*)*, the size of this subtree*
Choice(*L*) *is a function which selects in a uniformly random way an element of the set L*
ψ(n) is a function that gives the selection probability of a symbol if the subtree generated by the derivation of this symbol adds in the final tree more than n symbols compared to the size expected by the user
Result *– The created individual*

let R a tree with a root node A
while R contains a leaf which is a non terminal symbol
  let L the list of the leafs in the tree R
  let $f_{\min} := 0$
  for each symbol X of L
    $f_{\min} := f_{\min} + f(X)$
  end for
  let L_{nt} the list of the non terminal symbols of L
  let $T :=$ **Choice**(L_{nt})
  for each disjunction D_i of the right part of the rule T
    let $f_{add} := f(D_i)$
    let $f_{sub} := f(T)$
    $D_{i,\text{suppl}} := f_{\min} + f_{add} - f_{sub} - f_{ask}$
    $D_{i,\text{proba}} := \psi(D_{i,\text{suppl}})$
  end for
  Choose a disjunction $D \in \{D_1, ..., D_n\}$ of the rule T using the selection probabilities D_{proba}
  let R_{ins} a tree with a root node T and where each edges is one of the conjunctions of J
  Replace in the tree R the symbol T by the subtree R_{ins}
end while
Returns R, a tree with a root node A where the leafs are terminal symbols

Algorithm A3. Function TreeCreation

When a disjunction needs to be derived, a choice has to be made to select the best symbol B. This choice is related to the number of symbols needed to complete the current tree and the estimation of the number of symbols that can be generated by a derivation of B. Given a grammar G, a non terminal symbol B and a criterion F (F could be the expected size or height of a subtree generated by B), the algorithm uses a function $\psi(n)$ returning the selection probability of B if the subtree generated by the derivation of B adds in the final tree more than n symbols compared to the size expected by the user. It is clear that this probability should be low for high values of n. Several probabilistic functions for $\psi(n)$ (see algorithm A3) have been selected, among them:

$$\psi_1(n) = \frac{1}{a + |n|}, \tag{2}$$

$$\psi_2(n) = e^{-\frac{a \cdot n^2}{b}}, \tag{3}$$

where a and b are constants.

By experimenting on several regression problems, the function $\psi_2(n)$ was found to have good selection qualities using the constant values of $a = 2$ and $b = 5.2$.

4.4. Crossover operator

The proposed operator for crossing over two genetic trees is based on the following principles:

- the generated individuals must remain coherent at the output of the operator. In particular, the grammar rules must always be verified as well as the arities of the nodes (a terminal operator should never change its arity);
- the selection probability of each node must be identical. Even if the genetic recombinations actually implement a very complex chromosomal system, it is preferable to keep the same probabilities of mixing, as in the case of a simpler representation. For instance, a root node is not selected more frequently than a leaf node, so that the modifications are smoother;
- the algorithm must bring a guarantee that the resulting offsprings are different from the parents.

In conclusion, only subtrees coming from the same grammatical symbol are safe to be exchanged. The crossover operator, Algorithm A4, only deals with the genotypic description of the trees. The use of the genotypic trees guarantees the three points mentioned above. For instance, two subtrees, even deriving from the same terminal operator, will not be exchanged if they are not in the same grammatical *context*, i.e. if they derive from two distinct grammar rules. The resulting trees are converted into phenotypic trees and inserted in the new population.

Function Crossover **(A_1, A_2)**
Parameters – A_1 and A_2 are the two genotypic trees to cross
S(A) is a function which returns the list of the left-defined symbols of the grammar G defined in the tree A
Deriv(A,X) is a function which returns the list of the nodes in the tree A containing the symbol X
Choice(L) is a function which choose in a uniformly random way an element of the set L
GetChild(A, n) is a function which returns the child number n of the node A
X is a symbol of the grammar
n_1 and n_2 are nodes from genotypic trees
Results *– A'_1 and A'_2 are the two resulting genotypic trees*

for each tree $A \in \{A_1, A_2\}$
  while **CountChild**(A) = 1
    A := **GetChild**(A, 1)
  end while
end for
let $L := S(A_1) \cap S(A_2)$
let X := **Choice**(L)
let N_1 := **Deriv**(A_1, X)
let N_2 := **Deriv**(A_2, X)
let n_1 := **Choice**(N_1)
let n_2 := **Choice**(N_2)
Exchange n_1 and n_2 in the trees A_1 and A_2
Returns the resulting trees A'_1 and A'_2

Algorithm A4. Function crossover

Two remarks can be stated concerning Algorithm A4. First, the only case in which the offsprings would be identical to the parents is the case of a filiform genotypic tree. That corresponds to a grammar containing only one terminal symbol, consequently the corresponding phenotypic tree consists of only one symbol. In this case, the crossover cannot do better than exchange this symbol with one of the corresponding symbols of the other parent respecting the grammar. Second, the algorithm uses a parameter constraining the size of the generated trees. After the crossover, it is possible that this criterion is not respected any more. So, this criterion is verified *a posteriori* in the evaluation function.

The crossover parameters set by the user (besides the grammar) are the following:

- the percentage Q of individuals to be crossed. Most of the literature [Goldberg 1989; Schoenauer, Michalewicz 1997] consider that 80% is an acceptable value, but we obtained good results with a comprised value between 70 and 95%,
- the crossover selection type (roulette wheel, tournament, etc.).

4.5. Mutation operator

The mutation operator contains two significant characteristics; it imposes fewer parameterizations by the user and it preserves almost all the material from the parent. This has led us to define three different and complementary sub-operators applied in a uniformly random way. Each one of these operators takes as input a genotypic tree and returns the modified tree. The conversion into a phenotypic tree is necessary before the insertion into the new population for the calculation of the evaluation function. In all cases, the constraints (which have been explained earlier) for the crossover operator have to be also respected (coherence, probability of selection, production of new offsprings).

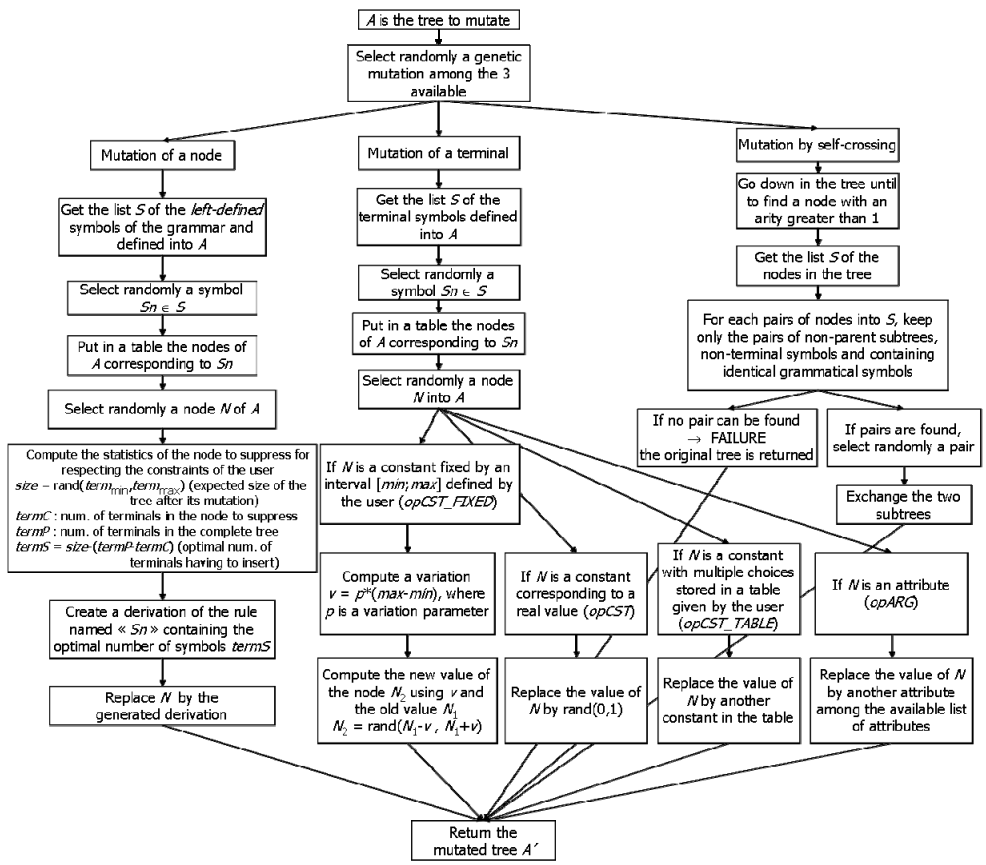

Figure 5. Mutation operators in GramGen. Note that rand(*a, b*) is a function returning a random value between *a* and *b*

The three sub-operators are as follows:

- Mutation of a node: performed by removing one of the nodes in the tree and replacing it by an equivalent one generated by the grammar.
- Mutation of a terminal: performed by changing one of the values of a numerical terminal (a constant or an argument).
- Mutation by self-crossing: performed by crossing the tree with itself. This sub--operator is complementary to the other ones because it can, for instance, reverse the numerator and the denominator of a ratio, which is not possible with the other sub-operators.

For the mutation operator (Figure 5), the user has to define the following parameters:

- the percentage Q of individuals to be mutated (the considered percentages have been set between 5 and 45%,
- the mutation type,
- and in the case of the terminal mutation operator, the new value is selected in an interval $[x - v; x + v]$ where x is the previous value and v is a parameter of variation related to the size of the authorized range for this value (so, this operator is data-scale independent).

To summarize, the described operators guarantee that the numerical constraints imposed by the user are respected (for instance, the size of a tree) as well as the user defined grammar. Some specific checks can be implemented to deal with the case of badly conceived grammars. In the case of the last mutation sub-operator (mutation by self-crossing), the exchange of a node with one of the progenitors of this node should never be permitted. However, this can occur with a grammar generating filiform trees. In this situation, the case is detected and the non-modified tree is returned.

5. Case study – function COSLOG

The goal of the experiment is to test our algorithm on a very simple regression problem, called the COSLOG problem. The algorithm has to interpolate a set of points given by the function $f(x) = \cos(\log(x))$. This function is very complex compared to the terminal operators available for this algorithm. The set of available terminals contains $(+, -, \times, /, |x|)$. The chosen grammar allows generation of trees with any number of nodes, but the *ideal number of nodes* parameter has been set to 10. In this case, trees of any sizes can be generated, but the fitness function will penalize trees that are too small or too large. This will give clues about the ability of the algorithm to model complex real world functions with only a limited set of operators. The experiment has been made more complex by limiting the number of available points: the training set contained only 20 pairs in the form $(x, f(x))$.

The learning has been carried out using the set of parameters shown in Table 1.

Table 1. Used parameters for the COSLOG problem

Parameter	Value
Sampling	20 instances
Population size	|P| = 50 individuals
Size of the trees	5 to 10 nodes
Termination criterion	50 generations
Operator set	{+, –, *, /, absolute value (*opABS*)}
Terminals	{constants (*opCST*), variable X (*opARG*)}
P_{mut}	0.40
P_{cross}	0.70
Selection operator	Direct ranking
Replacing operator	Direct ranking
Number of offspring per generation	|P|
Elitism	1% (high)
Duration	2 min (2.5 GHz CPU) for 20 instances

The grammar which has been used is this one:
Start symbol
$S \rightarrow E$

Definition of the core of the equation
$E \rightarrow O2\ E\ E \mid O1\ E \mid V$

Definition the operators (with one or two arguments)
$O1 \rightarrow opABS$
$O2 \rightarrow opADD \mid opSUB \mid opMUL \mid opDIV$

Definition of the terminals
$V \rightarrow opCST \mid opARG$

Figure 6 shows the COSLOG problem. The left part shows the function $f(x)$ for a training set of 20 points. More points have been sampled in the interval (0; 0.2] because of the shape of the function. The right part shows the function $f(x)$ (dashed line) and the function found by **GramGen** (dotted line). A horizontal logarithmic scale has been used in the second figure to facilitate visualization.

Equation 4 represents the formula which was finally generated and Equation 5 is its simplified form.

$$f_{GG}(x) = \frac{(0.316 + 0.833) \cdot |x|}{0.833 - 0.741 + x} - (0.316 \cdot 0.316 \cdot 0.741 \cdot 0.741 \cdot x)\,, \tag{4}$$

$$f_{GG}(x) = \frac{1.149 \cdot x}{0.091 + x} - (0.055 \cdot x)\,. \tag{5}$$

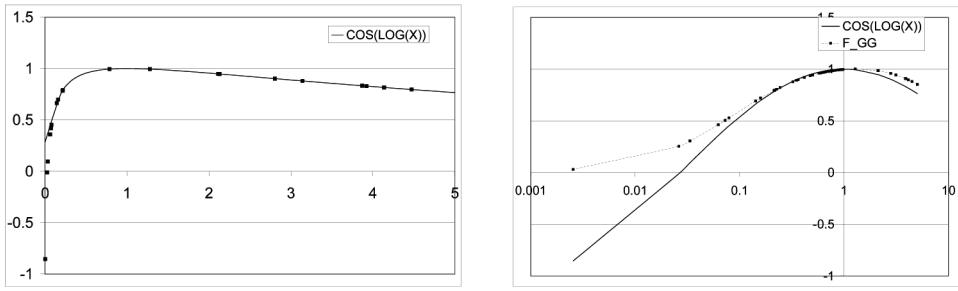

Figure 6. The COSLOG problem. Left part: function $f(x) = \cos(\log(x))$ and the training set. Right part: the function $f(x)$ (dashed line) and the function found by the algorithm (dotted line)

Discussion: A high correlation has been observed between the function interpolated and the obtained formula. The genetic evaluation of the individual corresponding to $f_{GG}(x)$ is 0.946 and the correlation coefficient between $f(x)$ and $f_{GG}(x)$ is 0.926. According to Figure 6, this correlation is also relatively high for small (~0.1) and high (~5) values of x, despite the relative simplicity of the obtained formula. Consequently, **GramGen** achieves a good performance for the small data set and for the complex function shown in this experiment.

GramGen has been extensively tested in classification of remote sensing images but this domain is out of the scope of this volume. Interested reader may find more information about this application in [Quirin 2005; TIDE 2005].

6. Conclusion

In this paper, a new approach to discover rules able to solve symbolic regression problems by grammar-based GP is proposed. In general, algorithms generating trees allow the user accessing a powerful representation, but are sometimes difficult to understand. This representation requires the redefinition of the genetic operators in such way that the generated individuals are coherent, according to the grammar. In the paper, specific attention has been given to the legibility and the complexity of the trees by integrating thresholds defined by the user. The thresholds control the number of nodes, the height of the trees and the expected accuracy using a small number of parameters.

Some tests have been carried out with this new approach using a user-defined grammar. On the tested problem, the obtained results have been acceptable in terms of accuracy on the testing set. Concerning the comprehensibility, the question of knowing why a tree is more comprehensible only because it is generated by a grammar written by an expert remains an interesting research perspective. In many fields, experts are still accustomed to precise schemata, and it is advisable to respect this predisposition.

In spite of the complexity of the mutation operator, the bulk of the computation is led in an automatic way to use the grammar which discharges the user from parameter setting. Nevertheless, this operator can be improved. For instance, the principle of the self-adapting mutation [Back, Hoffmeister, Schwefel 1991] has not been tested yet within the framework of **GramGen**.

The presence of constants involves a considerable increase of the size of this search space. An interesting topic for future work would be to optimize this search. Techniques in which some constants values are freezed and locally optimized have been described in the literature, but they have not been yet implemented in **GramGen**. During the experiments, the amount of use of the *opCST* operator was limited, which restrains the effect of over-fitting and returns formulas which are slightly more adapted to the new data. Another interesting point concerns the anticipated algebraic simplification of the obtained formulas during their evolution, either to reduce the search space, or to deliver more understandable formulas. Research work in these areas is needed and they will undoubtedly be considered as future research directions.

References

Back T., Hoffmeister F., Schwefel H. (1991), A survey of evolution strategies, [in:] *Proceedings of the 4th International Conference on Genetic Algorithms*, Eds. R.K. Belew, L.B. Booker, Morgan Kaufmann, San Diego, pp. 2-9.

Freeman J.J. (1998), A linear representation for GP using context free grammars, [in:] *Genetic Programming: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, pp. 72-77.

García-Arnau M., Manrique D., Ríos J., Rodríguez-Patón A. (2007), Initialization method for grammar-guided genetic programming, *Knowledge-Based Systems*, Vol. 20, No. 2, pp. 127-133.

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

Javed F., Bryant B., Crepinsek M., Mernik M., Sprague A. (2004), Context-free grammar induction using genetic programming, [in:] *Proceedings of the 42nd Annual ACM Southeast Conference*, Eds. S.-M. Yoo, L.H. Etzkorn, ACM, Huntsville, pp. 404-405.

Koza J.R. (1992), *Genetic Programming*, The MIT Press/Bradford Books, Cambridge.

Manrique D., Marquez F., Rios J., Rodriguez-Paton A (2005), Grammar Based Crossover Operator in Genetic Programming, [in:] Artificial intelligence and knowledge engineering applications: A bio-inspired approach, First International Work-Conference on the Interplay Between Natural and Artificial Computation (IWINAC 2005), *Lecture Notes in Computer Science*, Vol. 3562/2005, Las Palmas, pp. 252-261.

Montana D.J. (1994), *Strongly typed genetic programming. Technical Report #7866*, Bolt Beranek and Newman, Cambridge.

Quirin A. (2005), *Discovery of classification rules by an evolutive approach: Application to remote sensing images*, PhD Thesis, Louis Pasteur University, Strasbourg [in French].

Ross B.J., Gualtieri A.G., Fueten F., Budkewitsch P. (2002), Hyperspectral image analysis using genetic programming, [in:] *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, pp. 1196-1203.

Schoenauer M., Michalewicz Z. (1997), Evolutionary computation, control and cybernetics, *Evolutionary Computation* [Special Issue], Vol. 26, No. 3, pp. 307-338.

TIDE (2005). Tidal Inlets Dynamics and Environment, Research Project Supported by the European Commission under the Fifth Framework Programme, contract n° EVK3-CT-2001-00064, at http://www.istitutoveneto.it/tide.

GRAMGEN: SYSTEM PROGRAMOWANIA GENETYCZNEGO OPARTY NA GRAMATYCE BEZKONTEKSTOWEJ

Streszczenie: w artykule przedstawiono platformę programowania genetycznego, zwaną GramGen, łączącą ideę algorytmów ewolucyjnych z gramatyką bezkontekstową (CFG). Zadaniem systemu jest generowanie formuł logicznych (drzew genotypowych), które po przekształceniu do drzew fenotypowych (formuł) są w stanie rozwiązać konkretny problem praktyczny. Jako ilustrację podejścia podano przykład wykorzystania GramGen do regresji symbolicznej funkcji CosLog. W artykule opisano szczegółowo główne algorytmy, operatory genetyczne, sposóby definiowania ograniczeń oraz "strojenia" parametrów systemu.