Chapter 1
Tree-Based Grammatical Evolution with Non-Encoding Nodes

Marina de la Cruz, Oscar Garnica, J. Manuel Velasco, Daniel Parra and J. Ignacio Hidalgo

Abstract Grammar-guided genetic programming is a type of genetic programming that uses grammatical rules to constrain the search space of the solutions. There are different representations of grammar-guided genetic programming, including grammatical evolution, structured grammatical evolution, and context-free-grammar genetic programming. Each representation introduces distinctive characteristics that shape the evolutionary process. In this chapter, we propose a new representation that uses a tree structure with non-encoding nodes for the individuals in the population; Tree-Based Grammatical Evolution with non-encoding Nodes. This representation increases the size and complexity of the individuals while performing a more exhaustive exploration of the solution space. Each node of the tree will have a set of children nodes and an associated number; this number determines which of the children nodes are used in decoding the solution and which are non-encoding nodes. Those nodes carry out a concomitant evolutionary process that may manifest on the phenotype after eventual mutation or crossover events. We investigate the performance of our proposal with a set of well-known benchmarks. Experimental results analyze the differences and similarities with other methods that also use grammars to guide the solutions and indicate the utility of the Tree-Based Grammatical Evolution with Non-Encoding Nodes.

Index Terms

Grammar-Guided Genetic Programming, Dynamically Structured Grammatical Evolution, Context-Free Grammar Genetic Programming, Grammatical Evolution

---

Marina de la Cruz, Oscar Garnica, J. Manuel Velasco, Daniel Parra and J. Ignacio Hidalgo
Department of Computer Architecture, Universidad Complutense de Madrid, Spain e-mail: absys@ucm.es

1.1 Introduction

Grammar-Guided Genetic Programming (GGGP) is a variant of genetic programming (GP) that employs a grammar to define the structural composition of individuals within the population. The use of grammars in the construction of the solutions allows the creation of individuals that contain expert knowledge for the specific problem and limits the accessible search space.

Different GGGP versions have been proposed, including Grammatical Evolution (GE) [7], which uses a linear representation of the genotype, Structured Grammatical Evolution (SGE) [10], which uses multiple linear representations, one for each rule of the grammar and, Context-Free Grammar GP (CFG-GP) [22], where the genotype is a derivation tree generated from the grammar. Each representation has a different effect on the evolutionary process, with its advantages and drawbacks.

1.1.1 Context-Free Grammars

Context-free grammars (CFG) are formal grammars defined by Chomsky [6] that have been applied in various contexts, including GP. Formally, a CFG is defined as a tuple $G = (N, T, S, P)$, where N corresponds to a set of non-terminal symbols, T is the set of terminal symbols, S is the rule that begins the grammar, called axiom, and P the set of productions for each of the rules. The productions adhere to the form $A ::= B$, with A belonging to N, and B any sequence of terminals and non-terminals belonging to N and T respectively ($B \in (N \cup T)*$). Each non-terminal can have multiple productions separated by the | operator. This syntax, known as the Backus-Naur Form, will be employed throughout the chapter to define the grammar used to generate solutions. We will illustrate the GGGP techniques described in this chapter with the grammar of Figure 1.1.

```
# Model expression
<expr> ::= <expr> <op> <expr> | <pre_op> <expr> | <var>

#Variables
<var> ::= x1 | x2 | x3 | x4

#Mathematical operators
<op>  ::= + | -
<pre_op> ::= log() | sin() | cos()
```

Fig. 1.1 Grammar used in BNF format.

This grammar allows the GP algorithm to construct mathematical expressions in a recursive manner. It permits the inclusion (or not) of four variables ($x1, x2, x3$ and

$x4$) as many times as necessary and operated ($+$ or $-$) or preoperated (by $\log, \sin,$ and cos).

1.1.2 Grammatical Evolution

Grammatical Evolution (GE) [7] is a linear representation of a GGGP. In GE, the genotype of the individuals within the population is structured as a list of numerical values with a predetermined size. These numerical values can be expressed either as integers or as 8-bit binary numbers. They are initially generated randomly within the range of 0 to a specified maximum, typically set to 128 or 256. These numbers are used in the decoding process aimed at deriving phenotypes from the genotype.

The decoding process starts with the interpretation of the first non-terminal symbol specified by the grammar. This interpretation uses the first number of the genotype to obtain a production from the grammar rule, performing the modulus operator on the number of productions associated with the rule. Subsequently, it traverses the symbols of the selected production. Terminals are appended directly to the evolving phenotype, while non-terminals prompt the recursive application of the decoding process with subsequent numbers from the genotype list. In essence, the decoding process generates a derivation tree, traversed in pre-order, whose structure is determined by the genotype within the language specified by the CFG. The leaves of the derivation tree collectively constitute the phenotype. The phenotype will be a mathematical expression in a symbolic regression problem, a derivation tree in a classification problem or a computer program in a classical GP problem.

An example of a possible genotype structure, which can be derived from the grammar in Figure 1.1, is illustrated in Equation (1.1). It generates the phenotype $x1 + \sin(x3)$.

$$[3,2,8,4,1,7,5,14] \xrightarrow{\text{decode}} x1 + \sin(x3) \tag{1.1}$$

GE has some issues derived from its representation. These are uncontrolled locality and redundancy in gene operations. The concept of low locality is the notion that small changes in the genotype will likewise induce subtle changes in the phenotype. However, in the case of GE, we find that this is only sometimes the case, as performing small changes in the genotype may yield a substantially altered derivation tree during the decoding process. Additionally, redundancy emerges as another significant issue within GE. Most of the changes performed on the genotype have been shown to not produce a change on the phenotype [19]. This phenomenon arises from the use of the modulus operator, as the same production can be obtained from different numbers, resulting in the same phenotype.

The main advantage of GE, in comparison to other methods, is the decoupling of the genotype from the phenotype. It allows one individual to be used in different contexts and with different grammar structures without having to make any modifications.

1.1.3 Structured Grammatical Evolution

In lieu of GE's locality and redundancy problems, Lourenço et al. proposed a novel approach termed Structured Grammatical Evolution (SGE) [10]. SGE diverges from traditional GE in the genotype representation; instead of a list of numbers, SGE employs a list of lists, where each internal list corresponds to a non-terminal in the grammar. The decoding process is similar to GE, albeit with the incorporation of individual lists corresponding to each non-terminal symbol within the grammar. Decoding a specific non-terminal involves first, selecting the list that represents that symbol, second selecting the next unused number from that list, and finally, the number is used to get the next production from the grammar. The adoption of separate lists for each rule nullifies the need to use the modulus operator. In each internal list, the numbers are between 0 and $n-1$, with n being the total number of productions of the non-terminal symbol in question.

An example of a possible genotype structure that can be derived from the grammar of Section 1.1.1 is illustrated in Equation (1.2). It generates the phenotype $x1+\sin(x3)$.

$$\left[\overbrace{[0,2,1,2]}^{\text{<expr>}},\ \overbrace{[0,2]}^{\text{<expr>}},\ \overbrace{[0]}^{\text{<op>}},\ \overbrace{[1]}^{\text{<pre\_op>}}\right] \xrightarrow{\text{decode}} x1+\sin(x3) \tag{1.2}$$

SGE eliminates the need for the modulus operator, alleviating the redundancy issues found in GE. Additionally, the internal lists facilitate crossover operations at a finer granularity, producing children individuals with phenotypes more akin to their parents. Two versions of SGE have been proposed: static [11] and dynamic [10]. In this discourse, our focus will be on dynamic SGE, characterized by the dynamic generation of internal lists for each non-terminal symbol. During the initialization phase, the lists are generated up to what the specific individual needs to complete the phenotype generation and set the fitness. Following mutation and crossover operations, if more elements are needed for a rule to complete the phenotype generation, they are appended at the end of the internal list. This dynamic list management strategy ensures that only valid individuals exist in the population. Since the lists are dynamic, a maximum tree depth is imposed to mitigate potential bloating issues when generating recursive productions.

1.1.4 Context-Free Grammar Genetic Programming

In tree-based GGGP, individuals are derivation trees whose creation is guided by the grammar. Context-free grammar Genetic Programming (CFG-GP) is a version that uses a CFG to create the derivation trees. It was proposed by Whigham [22], and then several versions were developed. In this chapter, we adopt the original implementation of CFG-GP. The tree comprises nodes, each containing a symbol and

pointers to child nodes that collectively compose the tree structure. The structure of the tree is dictated by the grammar file, with the root node corresponding to the first non-terminal symbol specified in the grammar. Subsequent child nodes are determined by the production chosen by the individual tree. In CFG-GP, there is a one-to-one relationship between phenotype and genotype. Decoding entails traversing the tree in a pre-order fashion, extracting the values of leaf nodes, which contain terminal symbols, and generating the phenotype.

Figure 1.2 presents the genotype that generates the phenotype $x1 + \sin(x3)$ using the grammar in Figure 1.1.

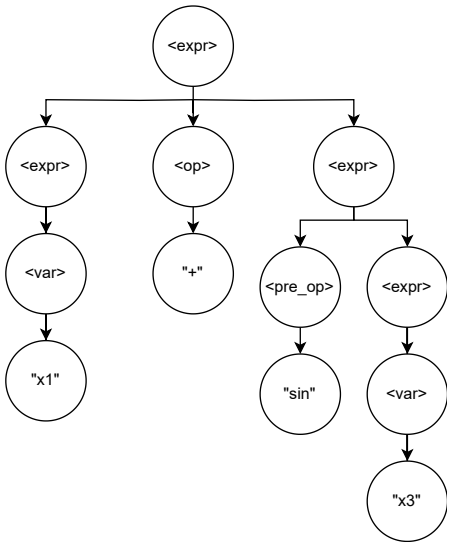

Fig. 1.2 Example genotype Context-Free Grammar GP

In CFG-GP, the derivation trees contain a possible structure that the grammar can generate by choosing a certain production for each non-terminal symbol. A random generation of trees might lead to an uneven population, particularly with complex grammars. Hence, different initialization techniques have been proposed to obtain a more uniform initial population [3, 8].

1.2 Tree-Based Grammatical Evolution with Non-Encoding Nodes (T-GE-NEN)

We propose a novel form of GGGP that combines features of CFG-GP and GE, namely Tree-Based Grammatical Evolution with Non-Encoding Nodes (T-GE-NEN). Central to our proposal is the use of non-encoding nodes, i.e., nodes that, while not directly impacting the phenotype, carry out a concomitant evolutionary process that

may manifest on the phenotype after eventual mutation or crossover events. These nodes preserve additional information about the grammar structure, so the evolutionary operators are not as destructive as other GGGP versions.

T-GE-NEN adopts a tree-based representation, where each node corresponds to a non-terminal symbol in the grammar. Algorithm 1 defines the attributes of each node: the non-terminal symbol governing the associated rule, a numerical identifier determining the selection of the rule's production, and the collection of potential child nodes representing subsequent non-terminal symbols.

Algorithm 1 Node definition

```
1: Node attributes
2:     symbol ← Symbol()
3:     production ← Integer()
4:     childrenNodes ← List[Node]
5: end Node attributes
```

In contrast to a traditional derivation tree, the set of children nodes may contain both encoding and non-encoding nodes. Whether a node is encoding is dependent on the production generated by its parent node. Figure 1.3 depicts a segment of a tree, exemplifying this concept.

Let us assume that the first digit in the genotype is 1. The upper node in the figure denotes the $< expr >$ non-terminal symbol, with the number 1 selecting the second production of the right-hand side of the rule associated with $< expr >$, producing $< expr >::=< pre\_op >< expr >$. Dividing this production into its non-terminal elements results in $< pre\_op >$ and $< expr >$. Both types of nodes will be present in the list of child nodes of the parent $< expr >$.

Being 1 the actual genotype, only $< pre\_op >< expr >$ is active in the current generation. However, if, during evolution, alterations occur in the numerical assignment associated with the parent (e.g., from 1 to 0), the production would become $< expr >::=< expr >< op >< expr >$.

This alternate production contains three non-terminal symbols. During the decoding process, T-GE-NEN would consider three children nodes: the first node of type $< expr >$, which is the same node that the previous production generated; the node of type $< op >$; and the second node of type $< expr >$. This configuration appears on the right-hand side of Figure 1.3.

One of the attributes of T-GE-NEN is that the nodes are shared among productions when they have the same type and amount of non-terminals.

1.2.1 Initialization

The initialization procedure for the data structure involves generating expansions, following the grammar structure, and constructing the tree accordingly. Algorithm 2

defines the procedure. This process closely resembles initialization for CFG-GP, with the inclusion of a numerical value assigned to all nodes to indicate the generated production. Furthermore, as shown in Figure 1.3, the terminal symbols do not generate nodes in the tree structure since the production number already represents them. The structure of all initial trees contains only encoding nodes; only the nodes used in the phenotype construction are generated. The rest of the structure will develop throughout the evolutionary process. The initialization process can be performed by generating random expansions until the entire tree has been generated or performing more complex generation techniques such as PTC2 [8, 12]. In Algorithm 2, line 3 parameterizes the generation of the next expansion.

Algorithm 2 Initialization process

```
1:  procedure INIT(node, depth, grammar, maxDepth)
2:      rule ← grammar.FINDRULE(node.symbol)
3:      node.production ← GENERATEPRODUCTION(rule)
4:      for all symbol ∈ rule.GET(node.production) do
5:          if symbol.ISNOTTERMINAL() then
6:              childrenNodes ← []
7:              production ← null
8:              newNode ← node.NEW(symbol, production, childrenNodes)
9:              INIT(newNode, depth+1, grammar, maxDepth)
10:         end if
11:     end for
12: end procedure
```

The basic structure is initiated by identifying the rule of the grammar associated with the symbol passed to the function. The next production is determined through the GENERATEPRODUCTION(*rule*) call. Afterwards, the production is set as the value of the current node. Then, the chosen production is iterated over, and all non-terminal symbols of the production create new nodes. Such nodes are generated and added to the set of children nodes of the parent through the *node*.NEW(*symbol,value, childrenNodes*) function where *node* is the parent node. The call creates a node whose type is the next non-terminal symbol of the production, with *childrenNodes*

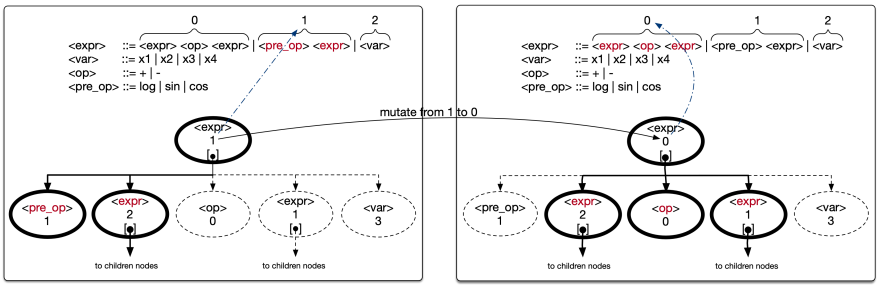

Fig. 1.3 T-GE-NEN mutation example

as an empty list of type *List[Node]* and an undefined value. This, initially, null number is modified recursively when the production number associated with the node is defined at GENERATEPRODUCTION. The function is called recursively with the new node, thus constructing the tree.

1.2.2 Decoding

Algorithm 3 describes the process used for transforming the genotype to the phenotype. The procedure assumes the dual task of validating the integrity of the genotype and compliance with the imposed size constraints. We have chosen to fuse both functionalities for efficiency reasons, so the number of times the tree is completely traversed is as small as possible. The decoding procedure obtains the grammar rule from the non-terminal symbol of the node and the production it generates. If the branch goes over the maximum depth, and the rule is recursive with the next production, evaluated with the function ISRECURSIVEPRODUCTION(*rule, production*), it modifies the production generated to become the shortest path to a terminal.

Algorithm 3 Decoding and revising procedure

```
 1: procedure DECODING(node, maxDepth, phenotype, grammar, depth)
 2:     rule ← grammar.FINDRULE(node.symbol)
 3:     production ← rule.GET(node.production)
 4:
 5:     if depth ≥ maxDepth && ISRECURSIVEPRODUCTION(rule, production) then
 6:         CHANGERECURSION(node, grammar)
 7:         production ← rule.GET(node.production)
 8:     end if
 9:     for all symbol ∈ production do
10:         if symbol.ISTERMINAL() then
11:             phenotype.APPEND(symbol)
12:         else
13:             nextNode ← node.NEXTCHILD(symbol)
14:             if nextNode == NULL then  ▷ Current node has no children nodes for the symbol
15:                 childrenNodes ← []
16:                 randomVal ← GENERATEPRODUCTION(symbol)
17:                 newNode ← node.NEW(symbol, randomVal, childrenNodes)
18:             end if
19:             DECODING(newNode, maxDepth, phenotype, grammar, depth+1)
20:         end if
21:     end for
22: end procedure
```

The algorithm then iterates over each symbol of the production. If the next symbol is terminal, it adds it to the construction phenotype. Otherwise, it retrieves the appropriate child node from a given grammar symbol through the procedure *node*.NEXTCHILD(*symbol*). This procedure should return the next unused child

node that contains the defined symbol. In cases where no unused node exists for the request, a null value is returned. In Algorithm 3, when it receives a null value instead of a valid node, the sub-tree is generated randomly, forcing the completion of the phenotype. This is performed through the GENERATEPRODUCTION(*symbol*) function, which returns a valid random value within the productions of the following rule *symbol*. The node.NEW(*symbol, randomVal, childrenNodes*) instruction has the same function as in the initialization procedure; it creates a new node and associates it with the parent. This may occur when, through the evolutionary process, a node changes the generated production, and the new production contains a symbol not present in the previous one. This is how the set of children nodes is generated dynamically through the evolutionary process. The depth restriction ensures that the newly generated nodes adhere to the depth limit set.

1.2.3 Crossover operator

Based on the structure of the genotype tree, the crossover operator is the general sub-tree crossover. The operator selects a non-terminal symbol in both parent solutions and exchanges the corresponding instances between the trees. Given the presence of both encoding and non-encoding nodes within the trees, specific constraints have been added to determine the type of nodes eligible for crossover. Algorithm 4 shows the constraints employed in our study. They dictate that 25% of the time, the selected node in both trees is encoding; 50% of the time, the node from one of the trees, selected at random, is chosen from all nodes, including the non-encoding ones; and the remaining 25% of the time the nodes from both trees are chosen randomly out of every now in the tree. Consequently, the probability of selecting a non-encoding node increases if the tree contains a higher density of these types of nodes. The constraints proposed are designed to ensure that at least 75% of the time, one of the final individuals undergoes a phenotype change. However, the specified rates are adjustable; for some problems, it may yield better results to allow more or less freedom in the node selection process. The procedure is described in Algorithm 4 by generating two random boolean values. Based on the results, a random encoding node may be retrieved, GETRANDOMENCODINGNODE(*ind, symbol*), or a random node from all available, GETRANDOMNODE(*ind, symbol*).

In line 18, the swap of the tree nodes includes all the children nodes, both encoding and non-encoding. Figure 1.4 presents an example of the described procedure. An alternate version of the crossover could consist of only the encoding children nodes of a sub-tree so that only the phenotypical part is swapped between the trees. The non-encoding nodes would remain in their original tree. This alternate version has been developed but will not be used in the results section.

Algorithm 4 Crossover procedure

```
1:  procedure SUBTREE CROSSOVER(ind1, ind2, grammar)
2:      bool1 ← UNIFORMRANDBOOLEAN()
3:      bool2 ← UNIFORMRANDBOOLEAN()
4:      symbol ← GETRANDOMSYMBOL(grammar)
5:      if bool1 && bool2 then
6:          node1 ← GETRANDOMENCODINGNODE(ind1, symbol)
7:          node2 ← GETRANDOMENCODINGNODE(ind2, symbol)
8:      else if bool1 then
9:          node1 ← GETRANDOMENCODINGNODE(ind1, symbol)
10:         node2 ← GETRANDOMNODE(ind2, symbol)
11:     else if bool2 then
12:         node1 ← GETRANDOMNODE(ind1, symbol)
13:         node2 ← GETRANDOMENCODINGNODE(ind2, symbol)
14:     else
15:         node1 ← GETRANDOMNODE(ind1, symbol)
16:         node2 ← GETRANDOMNODE(ind2, symbol)
17:     end if
18:     (node1, node2) ← (node2, node1)
19: end procedure
```

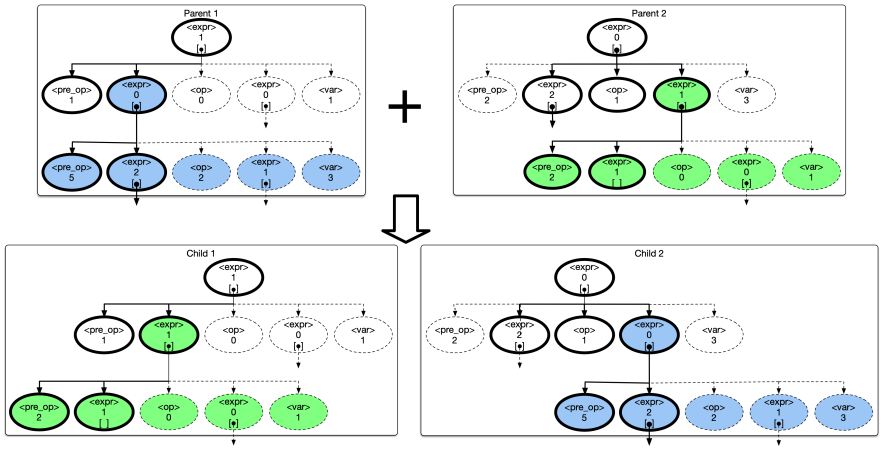

Fig. 1.4 T-GE-NEN crossover example. Dotted line circles represent non-encoding nodes, while solid-lined circles are the active (encoding) nodes. Green and blue colours highlight the parts of each parent participating in the crossover operation.

1.2.4 Mutation Operator

Mutation targets the attribute that determines the production encoded by each node. If a node is selected for mutation, a random number between 0 and the total number of productions of the non-terminal symbol will be drawn. The value of the node is swapped by the newly drawn number.

Similar to the approach adopted in the crossover, a set of constraints is incorporated into the mutation operator to account for the underlying structure. The selection procedure to determine which nodes are fit for mutation is divided between choosing from the encoding nodes and choosing from every tree node. A random value is drawn; if it is below a specific threshold, the node is chosen from the encoding nodes, or else it is drawn from every node of the tree, at which point the probability of it being encoded depends on the percentage of encoding vs non-encoding nodes. The threshold chosen for the study is set to 75%.

1.2.5 Bloating control

In T-GE-NEN, the issue of bloating is a significant concern, requiring control measures at both the encoding and non-encoding nodes. These measures can be performed independently. That is, a maximum tree depth can be established for the encoding nodes and a different tree depth assigned for the non-encoding nodes, provided that the overall tree sizes remain within a reasonable size. For the encoding portion, the control is performed through Algorithm 3.

The non-encoding nodes, on the other hand, are limited using a very similar technique, shown in Algorithm 5. The bloating control is performed at the end of each generation; once the individuals have gone through mutation and crossover, it is performed through a depth limit. If a tree branch goes over the depth limit and the rule symbol is recursive, then all recursive children's nodes are deleted. This is performed through the DELETERECURSIVENODES(*node, rule*) instruction, which simply deletes any children nodes of recursive nodes. If the next production the rule generates is recursive with the rule, ISRECURSIVE(*rule, production*), the production is modified to generate the shortest path to a terminal.

Algorithm 5 Bloating control for non-encoding nodes

```
 1: procedure CUTTREE(node, maxDepth, grammar, depth)
 2:     rule ← grammar.FINDRULE(node.symbol)
 3:     production ← nextRule.GET(node.production)
 4:
 5:     if depth ≥ maxDepth && ISRECURSIVE(rule) then
 6:         if ISRECURSIVEPRODUCTION(rule, production) then
 7:             CHANGERECURSION(node, rule)
 8:         end if
 9:         DELETERECURSIVENODES(node, rule)
10:     end if
11:     for all childNode ∈ node.childrenNodes do
12:         CUTTREE(childNode, maxDepth, grammar, depth+1)
13:     end for
14: end procedure
```

1.2.6 Other approaches with non-encoding genes

We can find in the literature some previous GP approaches where non-encoding genes participate in the evolutionary process. For example, stack-based genetic programming [15]. Perkins proposed a GP approach that utilizes an execution model that uses a stack-based virtual machine to evaluate GP individuals. Non-encoding nodes appear when the stack does not have enough values to perform an operation. For example, if an addition of two variables is indicated by the phenotype, but there is only one value on the stack. This is not a desired effect and produces non-valid individuals, so it is not comparable with the proposal presented in this chapter.

Non-encoding genes could also appear in Cartesian GP [13] under some conditions since not all genes in the genotype have to be active to decode a complete expression. As in T-GE-NEN, those genes pass genetic information through generations and genetic operators act on them too, experimenting a parallel *hidden evolution*. We explored also this alternative over hardware implementations [5]. As CGP uses a $\mu + \lambda$ evolutionary strategy algorithm [2], future work exploring T-GE-NEN under this selection scheme could be interesting.

1.3 Experimental Results

This section presents the methodology and experimental results obtained for the proposed algorithm.

1.3.1 Experimental Setup

Table 1.1 presents the experimental setup for conducting the tests. Parameters have been selected to ensure comparability across all algorithms, with values that work well on all of them. However, no studies have been performed on the efficiency of choosing specific parameters in relation to others other than the experimental results. Common to all of the algorithms is the number of generations, set to 1000, the size of the population, 200, and the number of runs, 100, for a total of 200000 evaluations. The selection and replacement methods, with tournament selection of 5 individuals and a replacement policy of the parents for the offspring with 2% elitism. PTC2 initialization is employed across all four algorithms.

In our GE implementation, the structure of individuals maintains a fixed size throughout the evolutionary process, in contrast to the other three implementations where the structure can grow. Although this characteristic reduces computation time, it also limits the generation process. To partially alleviate this issue, the initial size of the individuals and their expansions are doubled from 100 to 200.

Table 1.1 Experimental setup for comparison of GE, DSGE, CFG-GP and T-GE-NEN. SubTree* and One gene* are the evolutionary operators previously described

	GE	**DSGE**	**CFG-GP**	**T-GE-NEN**
Nº runs	100			
Gen	1000			
Pop	200			
Initialization method	PTC2			
Max expansions	200	100	100	100
Selection	Tournament pressure 5			
Elitism	2%			
Crossover operator	Single Point	Uniform (mask)	SubTree	SubTree* 50%
Crossover prob	90%			
Mutation operator	Integer-Flip	Integer-Flip	SubTree regeneration	One-gene* 75%
Mutation prob	1%	1%	100%	100%
Max Initial Depth	10	10	10	10
Max Depth	-	10	10	10
Chromosome length	200 (2 Wraps)	-	-	-

1.3.2 Benchmark Problems

Table 1.2 contains the list of benchmark problems selected to perform the tests. Most of the problems have been taken from [23]. They are four real-world problems and four mathematical functions. Six are regression problems, while the other two are classification problems.

Figure 1.5 depicts the grammar structure employed for all regression problems. The non-terminal $< var >$ is modified to contain the available variables specific to each problem. Correspondingly, Figure 1.6 illustrates the grammar structure used on the classification problems. The "`else if`" text is exclusive to the multi-class problem. For binary classification, the grammar only contains the first "`if`" statement. *RMSE* is the fitness function used for the regression benchmarks, whereas $1 - \text{Accuracy}$ is the fitness function of the classification benchmarks. The proposed grammars have been constructed to have a simple structure, especially on regression. Within the regression grammar, the sole recursive rule is $< expr >$, which, along with $< term >$, can generate non-encoding nodes in T-GE-NEN. In the classification grammar, both $< expr >$ and $< binexpr >$ are recursive.

Table 1.2 Benchmark problems

Name	Variables	Equation	Type	Training
Keijzer-6 [9]	1	$\sum_{i}^{x} \frac{1}{i}$	Regression	E[1, 50, 1]
Vladislavleva-4 [21]	5	$\frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	Regression	U[0.05, 6.05, 1024]
Nguyen-7 [20]	1	$\log(x+1)+\log(x^2+1)$	Regression	U[-1, 5, 1000]
Pagie Polynomial [14]	2	$\frac{1}{1+x^{-4}}+\frac{1}{1+y^{-4}}$	Regression	E[-5, 5, 0.4]
Dow Chemical (normalized) [17]	57	-	Regression	747 points
Abalone [24] [1]	8	-	Classification (3 classes)	2506 points
Banknote [16]	4	-	Classification (2 classes)	1000 points
Tower (normalized) [18]	5	-	Regression	4071 points

```
# Model expression
<expr> ::= <expr> <op> <expr> | <pre_op> <expr> | <var> | <term>

# options for terminals
<term> ::= <var> |<number>

# Variables
<var> ::= x1 | x2 | x3 | ..... xn

# Mathematical operators
<op>  ::= + | - | * | \
<pre_op> ::= log() | sin() | cos()| exp()| inv()| sqrt()| tan()

# Constants generation
<number> ::= <base> pow(10,<sign><exponent>)
<base> ::= 1|2|3|4| ..... |99
<exponent> ::= 1|2|3|4|5|6|8|9
<sign> ::= +|-
```

Fig. 1.5 Grammar for regression used in all representations.

1.3.3 Results

Table 1.3 displays the Mean, Median, Std. Dev and Min Value obtained during the training phase across all four algorithms. For all eight problem instances, the Mean values attained during training by T-GE-NEN were the best. The Median obtains lower results in all problems except for Nguyen-7. T-GE-NEN obtains the best overall result for six out of the eight benchmarks, with CFG-GP finding a lower value

```
# Model expression
<type> ::= if( <binexpr> ){class=0:}else{class=1:}
           else if( <binexpr> ){class=2:}
           ......
           else( <binexpr> ){class=n:}
# options for binary expressions
<binexpr> ::= ( <expr> <relop> <expr> ) |
              ( ( <expr> <relop> <expr> ) <binop> <binexpr> )

# Binary operators
<binop> ::= && | "||"
<relop> ::= "<" | ">" | "<=" | ">="

# Mathematical expressions
<expr> ::= ( <expr> <op> <expr> ) | <term> | <pre_op> <expr>

# Mathematical operators
<op> ::= + | - | * | \
<pre_op> ::= log() | sin() | cos()| exp()| inv()| sqrt()| tan()

# Variables
<var> ::= x1 |x2 |x3|...|xn

<term> ::= <var> |<number>
<var> ::= x1 | x2 | x3 | ... |xn
<op>  ::= + | - | * | \

# Constants generation
<number> ::= <base> . pow(10,<sign><exponent>)
<base> ::= 1|2|3|4|... |99
<exponent> ::= 1|2|3|4|5|6|8|9
<sign> ::= +|-
```

Fig. 1.6 Grammar for decoding classification models as if-then-else expressions .

on the Pagie Polynomial and the Dow Chemical problem. In general, the algorithm proposed seems to find better training results than the other alternatives.

Figure 1.7 illustrates the evolutionary trajectories observed across all benchmark instances. Each plot showcases the evolutionary progression of the best individual over 100 runs conducted for each benchmark. The solid line depicts the average fitness value of the best individual across each generation, while the error band is the standard deviation of the best fitness values across all runs for each generation.

From these graphs, we can observe that, on average, T-GE-NEN consistently outperforms the other three algorithms throughout the evolutionary process. CFG-GP emerges as the next best-performing algorithm, followed by DSGE and GE. However, the disparity in performance between CFG-GP and T-GE-NEN does not seem too pronounced compared to the other two algorithms. This observation can be explained by the fact that both CFG-GP and T-GE-NEN use tree structures for the evolution procedures. This type of structure is more complex, requiring greater

Table 1.3 Mean, Median, Standard deviation, and Best Fitness for each algorithm and problem. Bold values represent the lowest value

Benchmark	Algorithm	Mean	Median	Std. Dev	Min value
Pagie Polynomial	GE	0.391896	0.397145	0.093900	0.114032
Pagie Polynomial	DSGE	0.335472	0.330249	0.089490	0.082655
Pagie Polynomial	CFG-GP	0.222315	0.226208	**0.076206**	**0.031809**
Pagie Polynomial	T-GE-NEN	**0.179255**	**0.186572**	0.077647	0.032372
Tower	GE	0.106245	0.105829	0.017980	0.070854
Tower	DSGE	0.091207	0.089568	0.015240	0.066624
Tower	CFG-GP	0.078324	0.074586	0.014007	0.060275
Tower	T-GE-NEN	**0.073374**	**0.070832**	**0.011196**	**0.060141**
Vladislavleva-4	GE	0.193689	0.199103	0.017843	0.104983
Vladislavleva-4	DSGE	0.189788	0.189679	0.012619	0.152360
Vladislavleva-4	CFG-GP	0.143740	0.145498	**0.027961**	0.051027
Vladislavleva-4	T-GE-NEN	**0.137415**	**0.143208**	0.029619	**0.044778**
Dow Chemical	GE	0.115894	0.116878	0.019253	0.076254
Dow Chemical	DSGE	0.100412	0.098416	0.016260	0.066806
Dow Chemical	CFG-GP	0.081781	0.078189	0.013207	**0.057188**
Dow Chemical	T-GE-NEN	**0.077076**	**0.074802**	**0.010935**	0.060224
Abalone	GE	0.436077	0.435954	0.006069	0.418196
Abalone	DSGE	0.438344	0.438148	0.005766	0.423783
Abalone	CFG-GP	0.423412	0.425978	0.011943	0.393057
Abalone	T-GE-NEN	**0.412678**	**0.414405**	**0.009245**	**0.389864**
Keijzer-6	GE	0.045235	0.022517	0.054573	0.000471
Keijzer-6	DSGE	0.020618	0.007095	0.039168	0.000085
Keijzer-6	CFG-GP	0.002916	0.000289	0.009309	**0.000009**
Keijzer-6	T-GE-NEN	**0.002273**	**0.000278**	**0.005629**	**0.000009**
Nguyen-7	GE	0.264429	0.240511	0.095078	0.079676
Nguyen-7	DSGE	0.147206	0.136581	0.045373	0.065965
Nguyen-7	CFG-GP	0.076082	**0.056020**	0.047846	0.016678
Nguyen-7	T-GE-NEN	**0.064248**	0.057789	**0.032874**	**0.013889**
Banknote	GE	0.025770	0.021000	0.023478	0.000000
Banknote	DSGE	0.020850	0.019000	0.016235	0.000000
Banknote	CFG-GP	0.006380	0.000000	0.010288	0.000000
Banknote	T-GE-NEN	**0.003010**	**0.000000**	**0.006583**	**0.000000**

computational resources and time, but the changes are always limited to the chosen portion of the tree. So, more granularity can be achieved, as observed in the results.

Regarding the best fitness, T-GE-NEN overtakes the other problems early in the evolutionary process, usually before generation 50. The Vladislavleva-4 problem is the only exception. On it, CFG-GP and T-GE-NEN exhibit comparable evolutionary trajectories until approximately generation 400, after which T-GE-NEN starts to yield visible superior performance.

On the Abalone problem, the proposed algorithm obtains significantly better results compared to the next best-performing CFG-GP. Additionally, the evolutionary curve for the Abalone problem exhibits a less pronounced elbow shape than the other benchmarks. We attribute this phenomenon to the complexity of the classification grammar, specifically for the multi-class problem.

Figure 1.8 and Figure 1.9 present the outcomes derived from two techniques based on Bayesian models for comparing the methodologies. The Bayesian models are based on the Plackett-Luce distribution [4]. We use a significance level alpha = 0.05, with 20 Monte Carlo chains and 4000 simulations. The best fitness results for the 100 runs of all algorithms are sent to the Bayesian models for a total of 800 data points. Figure 1.9 reveals a statistical significance in the results obtained for all four algorithms since there is no line connecting each of them, which would signify no statistical significance. Figure 1.8 depicts the probability of winning for each algorithm under this Bayesian model. On it, our proposal, T-GE-NEN, has a higher probability of being the best and has also slimmer bounds, making the result more robust.

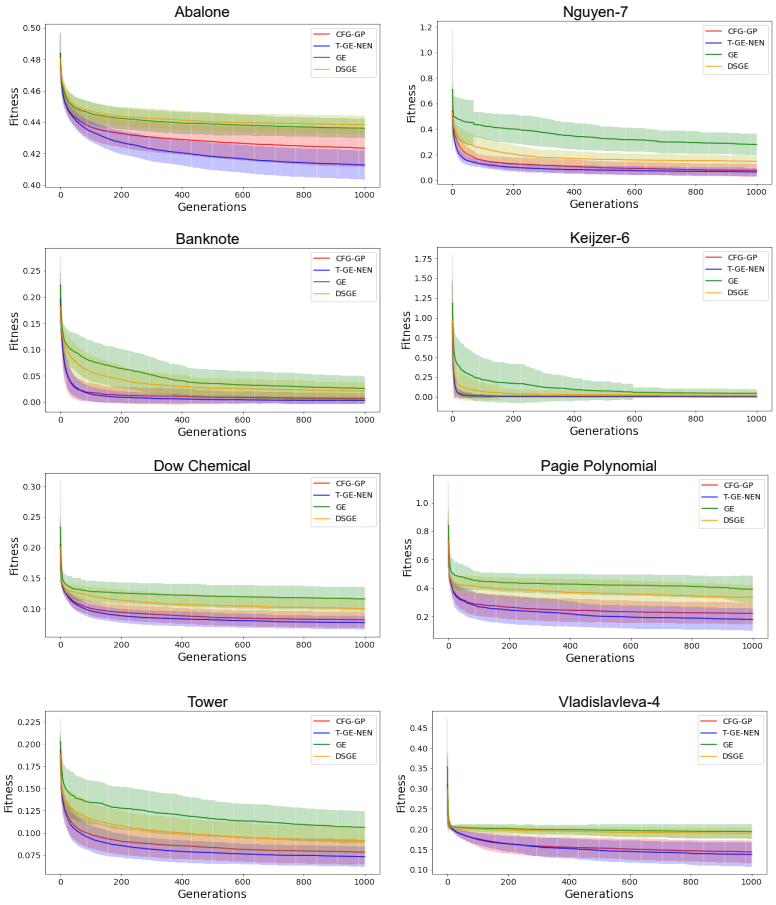

Fig. 1.7 Fitness generation evolution

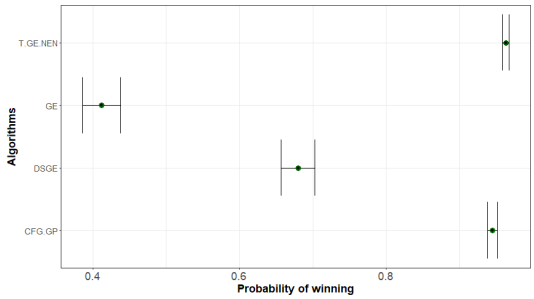

Fig. 1.8 Statistical analysis Bayesian ACC, Probability of winning

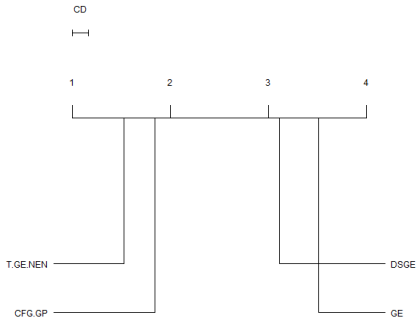

Fig. 1.9 Statistical analysis Bayesian CD ACC

1.4 Discussion

Several conclusions can be drawn from the presented results. Firstly, in the problems presented, the structure of the genotype clearly impacts the obtained results. Both tree representations, CFG-GP and T-GE-NEN yield better training values thanks to their more complex structures, which establish a one-to-one relationship between phenotype and genotype. Secondly, between T-GE-NEN and CFG-GP, our proposal, T-GE-NEN, obtained better results from the addition of non-encoding nodes to the tree structure. However, these non-encoding nodes increase the memory requirements of the genotype and increases the processing time for the evolutionary process, as computations must be performed on a larger number of nodes. The number of non-encoding nodes at any given time is mainly dependent on the tree depth and the grammar structure. More complex grammars, or those where productions don't share elements, will generate a greater number of non-encoding nodes. The amount

is exponential to the maximum tree depth. Consequently, our proposal is generally more computationally expensive than the other options presented.

1.5 Conclusions and Future Work

In this chapter, we have introduced T-GE-NEN, a novel variant of GGGP featuring tree-based individuals comprising encoding and non-encoding nodes, together with information on the production of the grammar in use. The initial results obtained for the algorithm are encouraging, yet further development and comprehensive investigation are warranted to realize its full potential.

In terms of future work, a more exhaustive study of how the different versions of the evolutionary operators perform across different problem complexities is essential. The same should be done in terms of the crossover and mutation probabilities of the different versions. From the point of view of the tree structure, the amount of non-encoding nodes will be dependent on the structure of the grammar, affecting the performance gain of T-GE-NEN versus a normal derivation tree. To test whether this algorithm truly improves the performance, a study is needed on different grammar structures for the same problem, with differing amounts of non-encoding nodes. Furthermore, the algorithm could be evaluated on other problem domains not considered in this chapter, such as program synthesis. The performance of the algorithm should also be assessed using test datasets.

Acknowledgements This work has been supported by The Spanish Ministerio de Innovación Ciencia y Universidad -grants PID2021-125549OB-I00 and PDC2022-133429-I00.

References

1. archive.ics.uci.edu: Abalone dataset. `https://archive.ics.uci.edu/dataset/1/abalone`. Accessed: 2024-03-10
2. Beyer, H.G., Schwefel, H.P.: Evolution strategies–a comprehensive introduction. Natural computing **1**, 3–52 (2002)
3. Böhm, W., Geyer-Schulz, A.: Exact uniform initialization for genetic programming. In: R.K. Belew, M.D. Vose (eds.) FOGA, pp. 379–407. Morgan Kaufmann (1996). URL `http://dblp.uni-trier.de/db/conf/foga/foga1996.html#BohmG96`
4. Calvo, B., Santafé, G.: scmamp: Statistical comparison of multiple algorithms in multiple problems. R J. **8**, 1–8 (2016). DOI 10.32614/RJ-2016-017
5. Cano, J., Hidalgo, J.I., Garnica, Ó.: Hardware real-time individualised blood glucose predictor generator based on grammars and cartesian genetic programming. Preprint at Research square (2024). DOI https://doi.org/10.21203/rs.3.rs-4262636/v1
6. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages**this work was supported in part by the u.s. army signal corps, the air force office of scientific research, and the office of naval research; and in part by the national science foundation; and in part by a grant from the commonwealth fund. In: P. Braffort, D. Hirschberg (eds.) Computer Programming and Formal Systems, *Studies in Logic and the Foundations of Mathe-*

matics, vol. 26, pp. 118–161. Elsevier (1959). DOI https://doi.org/10.1016/S0049-237X(09)70104-1. URL https://www.sciencedirect.com/science/article/pii/S0049237X09701041
7. Dempsey, I., O'Neill, M., Brabazon, A.: Grammatical Evolution, pp. 9–24. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). DOI 10.1007/978-3-642-00314-1_2. URL https://doi.org/10.1007/978-3-642-00314-1_2
8. García-Arnau, M., Manrique, D., Ríos, J., Rodríguez-Patón, A.: Initialization method for grammar-guided genetic programming. Knowledge-Based Systems **20**(2), 127–133 (2007). DOI https://doi.org/10.1016/j.knosys.2006.11.006. URL https://www.sciencedirect.com/science/article/pii/S0950705106001973. AI 2006
9. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Genetic Programming, pp. 70–82. Springer Berlin Heidelberg (2003). DOI 10.1007/3-540-36599-0_7
10. Lourenço, N., Assunção, F., Pereira, F., Costa, E., Machado, P.: Structured grammatical evolution: A dynamic approach, pp. 137–161. Springer International Publishing (2018). DOI 10.1007/978-3-319-78717-6_6
11. Lourenço, N., Pereira, F., Costa, E.: Sge: A structured representation for grammatical evolution. In: Artificial Evolution, pp. 136–148. Springer International Publishing (2015). DOI 10.1007/978-3-319-31471-6_11
12. Luke, S.: Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation **4**(3), 274–283 (2000). DOI 10.1109/4235.873237
13. Miller, J.F., et al.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proceedings of the genetic and evolutionary computation conference, vol. 2, pp. 1135–1142 (1999)
14. Pagie, L., Hogeweg, P.: Evolutionary Consequences of Coevolving Targets. Evolutionary Computation **5**(4), 401–418 (1997). DOI 10.1162/evco.1997.5.4.401. URL https://doi.org/10.1162/evco.1997.5.4.401
15. Perkis, T.: Stack-based genetic programming. In: Proceedings of the First IEEE conference on evolutionary computation. IEEE world congress on computational intelligence, pp. 148–153. IEEE (1994)
16. PonyGE: Banknote normalized dataset. https://github.com/PonyGE/PonyGE2/blob/master/datasets/Banknote/Train.csv (2017). Accessed: 2024-02-05
17. PonyGE: Dow chemical normalized dataset. https://github.com/PonyGE/PonyGE2/blob/master/datasets/DowNorm/Train.txt (2017). Accessed: 2023-02-10
18. PonyGE: Tower normalized dataset. https://github.com/PonyGE/PonyGE2/blob/master/datasets/TowerNorm/Train.txt (2017). Accessed: 2023-02-10
19. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Genetic Programming, vol. 3905, pp. 320–330. Springer Berlin Heidelberg (2006). DOI 10.1007/11729976_29
20. Uy, N.Q., Hoai, N.X., O'Neill, M., McKay, R.I., Galván-López, E.: Semantically-based crossover in genetic programming: application to real-valued symbolic regression. Genetic Programming and Evolvable Machines **12**(2), 91–119 (2011). DOI 10.1007/s10710-010-9121-2. URL https://doi.org/10.1007/s10710-010-9121-2
21. Vladislavleva, E.K., Smits, G., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. Evolutionary Computation, IEEE Transactions on **13**, 333 – 349 (2009). DOI 10.1109/TEVC.2008.926486
22. Whigham, P.: Grammatically-based genetic programming. In: Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications (1999)
23. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better gp benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013). DOI 10.1007/s10710-012-9177-2. URL https://doi.org/10.1007/s10710-012-9177-2
24. Wilkinson, L., Anand, A., Tuan, D.N.: Chirp: a new classifier based on composite hypercubes on iterated random projections. In: Proceedings of the 17th ACM SIGKDD International

Conference on Knowledge Discovery and Data Mining, KDD '11, p. 6–14. Association for Computing Machinery, New York, NY, USA (2011). DOI 10.1145/2020408.2020418. URL https://doi.org/10.1145/2020408.2020418