A String Representation Methodology to Generate Syntactically Valid Genetic Programs

SÓCRATES TORRES, MÓNICA LARRE, JOSÉ TORRES
Computer Science Department
ITESM Campus Cuernavaca.
Paseo de la Reforma 182-A Lomas de Cuernavaca 62589. Cuernavaca, Morelos.
MÉXICO

Abstract: - The need to generate not only syntactically valid Genetic Programs but also programs which remain syntactically valid, even after the applying of the crossover and the mutation operators is crucial for the good evolving of the genetic programs. In this sense, programs shall be represented such way that constants, variables and operators inherent to the problem be correctly represented during all the evolving process long. Actually, a string representation does not get the above requirements due to the syntactically wrong programs produced by genetic operators. This paper propose a String Representation Methodology to generate syntactically valid Genetic Programs. The Symbolic Regression is the area which is showed our representation.

Key-Words: - Genetic Programming, String Representation, Symbolic Regression.

1 Introduction

The Symbolic Regression tries to adjust an arithmetic expression to fit as much as possible the whole set of points that describe the behavior of a problem[1], the figure 1 shows an example of both a desired and an approximated curve. The more a curve is adjusted by the Symbolic Regression to the desired curve the better results are gotten when used this curve in order to classify new and unknown inputs or to predict outputs for unspecified inputs.

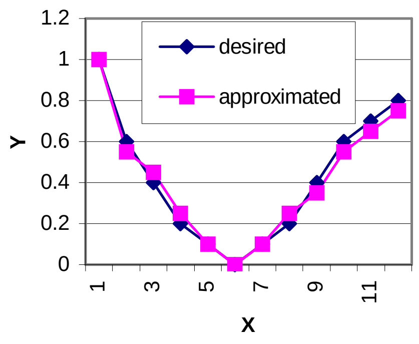

Fig. 1.- An example of both a desired curve and a Symbolic Regression approximated curve.

As is well known, at the beginning of a Genetic Programming Process, a genetic population of programs is randomly generated. A common problem is introduced in each genetic program when the arithmetic expression which describes the program possesses a non valid syntactical structure. Equation 1 shows a valid program while the equation 2 a non valid program.

$$VP = X + 1 * 4*X - 2/2*X \qquad (1)$$
$$NVP = X +/ 3 -* 5X +-* \qquad (2)$$

In this sense, we shall ensure that the method which generates the programs is leaded for a valid String Representation Methodology.

Now, suppose we have all of the program in the initial population having a right structure, the following step is to evaluate via the fitness function how good each program is to solve the current problem, the Symbolic Regression. Subsequently, in order to reproduce the programs into the next population, we may apply one of the Reproduction operators reported in most of the scientific papers related to the Genetic Programming area (e.g. Proportional Selection[2], Tournament[3]). Using any of the above reproduction methods does not introduce alteration in the structure of the program, so they maintain their validness.

However, applying the crossover operator may produce syntactically non valid programs. Consider the following valid programs, see the syntax's trees in the figures 2 and 3:

$$VP1 = X + 5 * X – 2* X + 1 \qquad (3)$$
$$VP2 = X * X –2 * 4 \qquad (4)$$

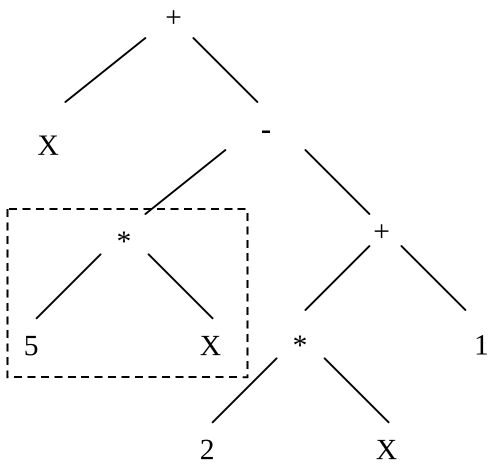

Fig. 2.- Syntax's tree for the valid program 1.

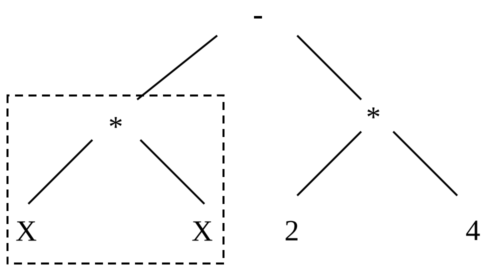

Fig. 3.- Syntax's tree for the valid program 2.

If crossover positions chosen randomly are element 4 for VP1 and element 3 for VP2 the resulting programs are:

Before crossover:

$$VP1 = X + 5\underline{\ * X – 2 * X + 1}$$
$$VP2 = X *\underline{\ X –2 * 4}$$

After crossover:

$$NVP1 = X + 5\ \underline{X – 2 * 4} \qquad (5)$$
$$NVP2 = X * \underline{* X – 2 * X + 1} \qquad (6)$$

As we can see, there is no operator between elements 3 and for in the equation 5 and a double operator between elements 1 and 4 in the equation 6. Then, the whole genetic process is affected due to syntax's defaults within the programs.

Would be good enough if we consider just homogeneous crossover points (e.g. operator-operator)? Consider now a crossover at the element 4 in equation 3 and the element 2 for equation 4, both of them are operators.

Before crossover:

$$VP1 = X + 5\underline{\ * X – 2 * X + 1}$$
$$VP2 = X\ \underline{* X –2 * 4}$$

After crossover:

$$NVP3 = X + 5\ \underline{* X –2 * 4} \qquad (5)$$
$$NVP4 = X\ \underline{\ * X – 2 * X + 1} \qquad (6)$$

It happen to be that we have gotten two valid programs, in fact they are. But our intentional crossover was not fulfilled, observe figures 2 and 3 where is shown the syntax's trees for both the VP1 and the VP3. The dotted squares shown in these figures illustrate two sub-trees, which according to [4], the planned crossover was a swapping between them. Said in other words, offspring programs should preserve most of the information contained in their parents [5].

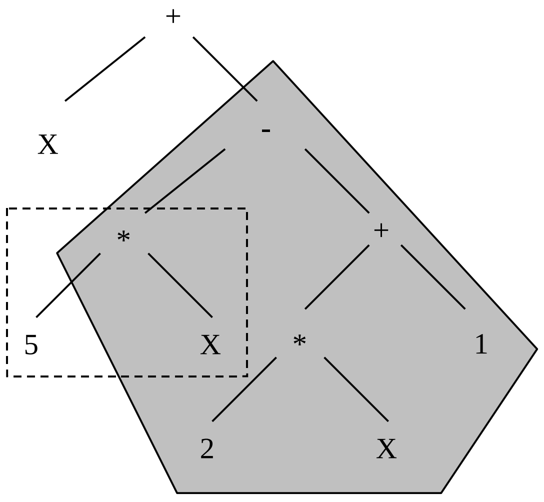

Fig. 4.- The Intentional and Real crossovers for program 1.

Now, if we consider the resulting programs in equations 5 and 6, Although they are structural valid are product of a very different crossover, see figures 4 and 5. While the intentional crossover is an exchange between sub-trees enclosed for the doted squares the real crossover was a big swapping between those “sub-trees” in the shaded vicinities. The resulting trees (offspring trees) are mostly the product of a randomly generated tree than a lengthening of the parents’ information.

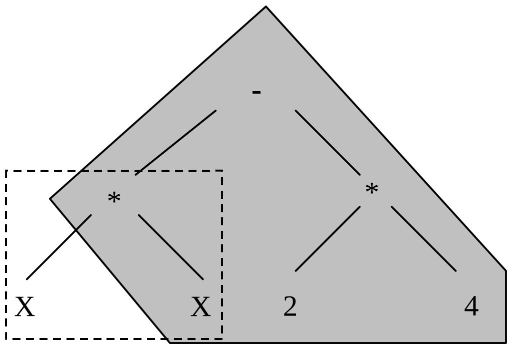

Fig. 5.- The Intentional and Real crossovers for program 2.

What happen if we consent this kind of erroneous crossover? What we can foresee is a random search process which will not be able to assess any good result.

The same problem is observed during the applying of the Mutation operator, consider the program in equation 3 and a Mutation at its element 4:

Before mutation:

$$VP1 = X + 5\underline{*}X - 2 * 10*X + 1$$

After mutation:

$$NVP5 = X + 5\underline{X}X - 2 * 10*X + 1 \qquad (7)$$

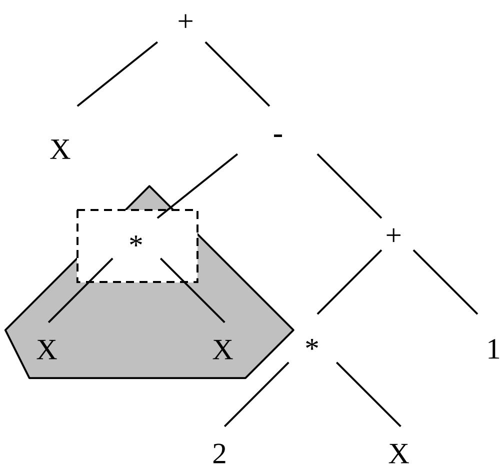

Fig. 6.- The Intentional and Real mutations for program 1.

Equation 7 shows a non valid program with three operands between elements 2 and 6. Even though we may formulate a special Mutation operator (e.g. If element equals operator then randomly generate another operator, else, generate either a variable or a constant), the specifications for a mutation say that we have to choose any element (either operator or variable or constant) from the program, eliminate de sub-tree rooted at this point and randomly generate another sub-tree [4]. The figure 6 shows a doted square with the real mutation and a shaded vicinity with the desired one.

The following sections describe the proposed String Representation along with the methodology to apply both the crossover and the mutation operator in such way that the structure of the programs remain correct. The problem related to the Symbolic Regression is used to show how this representation works.

2 Developed Methodology

As a problem formulation, we can declare that it is required a string representation which not only produces syntactically valid genetic programs but also that the structure of the programs remains accurate amid the genetic process.

2.1.- Generational Methodology

In order to adjust an arithmetic expression to a desired set of points, our representation must include two kind of elements: a set *NT* of non terminal elements (e.g. *+, -,*,/, abs( ), sin( ), pow( ),* etc.) and a set *T* of terminal elements (e.g. constants such as *4, 3* and *9* or variables such as *X* or *Y*).

Be *X* a vector of inputs $\{x_1, x_2, \ldots, x_n\}$ and *Y* a vector of desired outputs $\{y_1, y_2,\ldots,y_n\}$, *see figure 1, and* $D=\{1,2,\ldots,9\}$. We may consider the following sets $NT = \{+, *, -\}$ and $T = \{X, D\}$ with a numeric representation of NT ={-3, -2, -1} and T={0, [1,9]} to build any program within the initial pool and propose a recursive methodology to generate syntactically valid genetic programs; with the intention of avoid using parenthesis in the program, we have chosen a postfix notation to generate our programs:

```
programGeneration(NumberOfNonTerminals)
        non_terminalGeneration( )
        left_treeGeneration( )
        right_treeGeneration( )
        return
terminalGeneration( )
        counter = 1
        generate a random number from NT
        return
left_treeGeneration( )
   if (counter < NumberOfNonTerminals)
        generate either a
        random number from NT or T
        if (number_generated ε NT)
                counter++
                left_treeGeneration( )
                right_treeGeneration( )
   else
        generate a random number from T
   return
right_treeGeneration( )
         if( counter < NumberOfNonTerminals )
                generate a random number from NT
                count++
                left_treeGeneration( )
                right_treeGeneration( )
                return
        if ( counter = NumberOfNonTerminals)
                generate a random number from T
        return
```

VP0	<S5>	-2	-3	-3	-3	-2	3	7	3	2	5	4
VP1	<S5>	-2	0	-1	2	-3	5	-1	-2	9	7	6
VP2	<S5>	-2	4	-1	-1	-3	7	-1	1	0	0	0
VP3	<S5>	-1	-2	0	-3	0	–2	1	–1	3	0	8
VP4	<S5>	-1	-3	-2	0	-3	-3	5	0	6	0	0
VP5	<S5>	-1	-1	2	-3	-3	-3	1	1	7	5	3
VP6	<S5>	-2	9	-1	-2	0	-3	-2	6	5	4	6
VP7	<S5>	-2	-1	3	-3	2	-3	5	-1	9	8	0
VP8	<S5>	-2	-1	-3	-3	1	-1	9	4	0	2	0
VP9	<S5>	-1	-3	-3	-1	-2	7	0	0	0	1	3

Table 1.- Initial pool of syntactical valid Genetic Programs.

The table 1 illustrates a set of ten syntactical valid programs generated via the above methodology, each program has a size of *5* non-terminal elements (*S5*). The corresponding set of real programs are shown in the table 2.

As we can see, every program begins with a non-terminal (binary operator) followed by a left sub-tree and a right sub-tree. Every left sub-tree begins either with a terminal or a non-terminal element (depending on the non-terminals' counter), for each non-terminal element a recursive call is done to generate both a left sub-tree and a right sub-tree. Every right sub-tree mostly begins with a non-terminal element depending on the non-terminals' counter or with a terminal element.

VP0	<S5>	mul(add(add(add(mul(3,7),3),2),5),4)
VP1	<S5>	mul(X,sub(2,add(5,sub(mul(9,7),6))))
VP2	<S5>	mul(4,sub(sub(add(7,sub(1,X)),X),X))
VP3	<S5>	sub(mul(X,add(X,mul(1,sub(3,X)))),8)
VP4	<S5>	sub(add(mul(X,add(add(5,X),6)),X),X)
VP5	<S5>	sub(sub(2,add(add(add(1,1),7),5)),3)
VP6	<S5>	mul(9,sub(mul(X,add(mul(6,5),4)),6))
VP7	<S5>	mul(sub(3,add(2,add(5,sub(9,8)))),X)
VP8	<S5>	mul(sub(add(add(1,sub(9,4)),X),2),X)
VP9	<S5>	sub(add(add(sub(mul(7,X),X),X),1),3)

Table 2.- Real formatted Initial pool.

At this point, it was shown that the proposed string representation methodology generates a set of programs having a valid full syntactical structure.

2.2 Crossover Methodology

During the crossover process, we have to choose randomly both two programs and a crossover point for each program. Next, We have to extract the sub-trees rooted at the selected crossover points in order to swap them.

Consider the programs shown in the equations 3 and 4, if we hope to crossover them at the elements 4 for VP1 and 2 for VP2, see sub-trees enclosed for the dotted squares in the figures 2 and 3, we must construct 3 sub-string for each program as is illustrated in the figure 7.

Our partially recursive proposed methodology is:

```
Extract non recursively from VP1 into S1a
        The substring [0, crossover_point - 1]
Extract recursively from VP1 into S1b
        The sub-tree rooted at crossover_point
Extract non recursively from VP1 into S1c
        The ending string

Repeat extracting from  VP2 into S2
Swap S1b and S2b
End
```

The recursive procedure:

```
recursiveExtraction( )
    copy sub-root into string
    sub_treeExtraction( )
    sub_treeExtraction( )
    return
sub_treeExtraction( )
    if (element ε T)
        copy element into string
        return
    else
        copy sub-root into string
        left_treeExtraction( )
        right_treeExtraction( )
        return
```

Arithmetic notation:

*VP1 = X + 5 * X – 2 * X + 1*

*VP2 = X * X –2 * 4*

Numeric postfix notation:

VP1 = -3 0 –1 –2 5 0 –3 –2 2 0 1

VP2 = -1-2 0 0 –2 2 4

Recursive extraction process:

S1a=-3 0 –1, S1b=-2 5 0, S1c=–3 –2 2 0 1

S2a= -1, *S2b=*-2 0 0 *S3c=*–2 2 4

Resulting programs:

VP1 = -3 0 –1 –2 0 0 –3 –2 2 0 1

VP2 = -1-2 5 0 –2 2 4

Fig. 7.- Crossover process.

As we can see in the figure 7 and using the above methodology, 6 strings are extracted, two of them were extracted in a recursively way (the sub-trees that we must swap). After swapping them the crossover process is success full concluded.

2.3 Mutation Methodology

The methodology to get a right mutation may be derived from the crossover methodology. In fact, just one program and its mutation point are randomly chosen, then we apply *the extraction from VP into S* procedure to formulate the three strings (*Sa*, *Sb* and *Sc*):

```
Extract **non recursively** from VP into Sa
    The substring [0, mutation_point - 1]
Extract **recursively** from VP into Sb
    The sub-tree rooted at mutation_point
Extract **non recursively** from VP into Sc
    The ending string
Generate a new program P using
    The proposed generational methodology
Concatenate Sa + P + Sc as the new program
    Note: Sb is removed
End
```

The above methodology is able to use some of the procedures implemented in the proposed crossover methodology. The result is an accurate mutation.

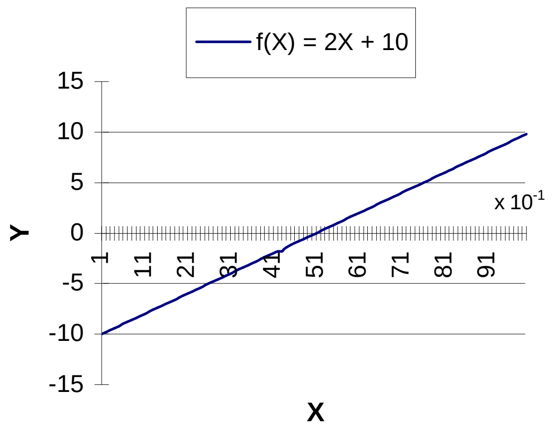

Fig. 8.- Set of desired points.

3 Experimental Results

We have used our proposed over all methodology in the Symbolic Regression field. Our goal is not to solve a big and hard problem but to prove that our *string representation methodology* is correct. This way, using the genetic process, we hope to get a program which fits the set of points showed in the figure 8. Figures 9, 10 and 11 show three different programs that asses the set of points.

As seen, program 1 is gotten at generations 11, program 2 at generation 7 and program 3 at generations 6. Two of them have dissimilar sizes.

Of course, we have considered the scalability of our methodology into not only more complex problems related to Symbolic Regression but also as a tool during the Automatic Text Decomposition and Structuring process, our main interest area.

Program 44 from Generation 11

Numeric postfix notation:
<S2> -1 0 –1 9.88 0
Real postfix notation:
<S2> sub(X , sub(9.88 , X))
Real infix arithmetic notation:
<S2> X – (9.88 - X)
simplified arithmetic expression:
<S2> ***2X + 9.88***

Fig. 9.- Generated program 44 of size 2 from generation 11.

Program 34 of Generation 7

Numeric postfix notation:
<S2> -1 0 –1 9.69 0
Real postfix notation:
<S2> sub(X , sub(9.69 , X))
Real infix arithmetic notation:
<S2> X – (9.69 - X)
simplified arithmetic expression:
<S2> ***2X + 9.69***

Fig. 10.- Generated program 34 of size 2 from generation 7

4 Conclusion

We have proposed a String Representation Methodology to generate not only syntactically valid Genetic Programs but also programs which remain syntactically valid, even after the applying of the crossover and the mutation operators. Such methodology is formed by: A generational methodology, a crossover methodology and a mutation methodology.

Furthermore, we have showed the validness of the programs our methodology generates applying it in the Symbolic Regression area.

Program 40 of Generation 6

Numeric postfix notation:
<S8> -1 –1 -1 0 –1 -8.19 -5.84 –1 -6.76 0 -3 –1 –1 -3.43 -6.35 -3.70 -7.21
Real postfix notation:
<S8> sub(sub(sub(X , sub(-8.19 , -5.84)) , sub(-6.76 , X)) , add(sub(sub(-3.43 , -6.35) , -3.70) , -7.21))
Real infix arithmetic notation:
<S8> {[X – (-8.19 – (-5.84))] –(-6.76 -X)} – {[(-3.43-(-3.43-(-6.35)))-(-3.70)] + (-7.21)}
Simplified arithmetic expression:
<S8> 2X + 9.70

Fig. 11.- Generated program 40 of size 8 from generation 6.

References:

[1] X1. Hans Gerber, Simple Symbolic Regression Using Genetic Programming, http://alphard.ethz.ch/gerber/approx/default.html last visited: January 15th, 2002.

[2] X2. D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley, 1989.

[3] X3. Zbigniew Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs. Springer Verlag , 1994.

[4] X4. J. R. Koza, F. H. B. III, D. Andre, and M. A. Kaene, Genetic Programming III. Morgan Kaufmann Publishers, 1999.

[5] X5. Scott Brave, What is Genetic Programming? http://www.genetic-rogramming.com/gpanimatedtutorial.html last visited: February 7th, 2002.