Automatic Generation of APL Programs

Manuel Alfonseca

Abstract

This paper describes the use of grammatical evolution to generate APL programs which perform some pre-required function.

Introduction

Genetic algorithms are optimization tools that simulate the principles of natural evolution and search for the minimum of an objective function. They operate on a population of chromosomes. A goal function (termed the *fitness function*) provides a mechanism to evaluate each element in the population. Starting from the current population, the next population is generated by means of several probabilistic genetic operators: selection, crossover, mutation, elision and fusion.

- Selection: The best elements in the population are selected according to their fitness. Solutions with better fitness have more chance to survive in the next generation.
- Crossover: this operator combines the genotypes of two members of the population to generate new members by exchanging parts of the parental genotypes.
- Mutation: some components in the progeny genotypes are modified randomly.
- Elision: some components in the progeny genotype are randomly deleted.
- Fusion: some components in the progeny genotype are randomly replicated.

Evolutionary programming is a computer science branch whose goal is the automatic generation of computer programs that solve a particular problem or perform a given pre-defined function.

The first approach to evolutionary programming [Koza 1992] was genetic programming (GP), originated in the nineties, which uses genetic algorithms to generate the desired programs, written in a selected computer language. This technique has proved to be an effective and efficient generic search and optimization method in a wide variety of situations, but presents an important problem: the syntactic correction of the generated programs is not guaranteed.

A more modern technique [ONeill 2001], grammatical evolution (GE), has been elegantly applied to automatic programming [O'Neill and Conor 2003]. This is a special case of genetic algorithms whose similarity to biological evolution is even greater, for a distinction is made between genotypes (strings of integers, the targets for the genetic operators) and phenotypes (programs written in a computer language). The translation between genotype and phenotype (the equivalent to embryo development in biology) is mediated by the grammar of the chosen computer language, usually described in Backus-Naur Form (BNF). In this way, the algorithms become independent of the language. One advantage of this method is the fact that all generated programs are syntactically correct (for they have been generated according to the grammar of the language). This allows GE to avoid one of the main difficulties in automatic evolutionary programming and reduces significantly the size of the search space.

Grammatical evolution

The following scheme shows the way in which GE combines traditional genetic algorithms with genotype to phenotype mapping:

1) Generate randomly an initial population of genotypes: strings of integers, usually with values between 0 and 255.
2) Translate each member of the current population of genotypes into the corresponding phenotype.
3) Compute the fitness of every phenotype in the population.
4) Sort the genotype population by the fitness of the genotypes.
5) If the best individual in the population is the required solution, the algorithm ends.
6) Replace the worst individuals in the population

by the genetically modified offspring of the best individuals. The four genetic operators mentioned above (crossover, mutation, elision and fusion) may be applied, although mutation is usually most effective for grammatical evolution.

7) Go to step 2.

Step 2, the translation of the integer genotypes into their equivalent phenotypes, is deterministically performed by means of the following process:

1) Initialize a string with the axiom (root non-terminal symbol) of the programming language grammar (usually <Program>, <Function>, or something similar).
2) Get the next integer number (*codon*) in the current genotype. If there are no more codons available, start again at the left end of the genotype. This biologically inspired wrapping mechanism is similar to the gene-overlapping phenomenon observed in many organisms in nature.
3) Choose the leftmost non-terminal symbol in the current string. Locate all the rules in the grammar whose left hand side is that symbol. Let their number be n. Number them in zero origin, in the order they appear in the grammar.
4) Select the rule to be applied as `n | codon`. This mapping introduces genetic code degeneracy, another biologically imitated feature, which means that different codons in the genotype generate the same phenotype.
5) Derive the next string by replacing the current non-terminal symbol by the right hand side of the selected rule.
6) If the new string contains no non-terminal symbols, the process is ended. Otherwise, go to step 2.

Automatic program generation in APL

I have used the grammatical evolution technique to automatically generate APL functions written in a subset of the APL2 language, by means of grammatical evolution, with the following modification:

- Rather than generating a fully general APL function, I decided to simplify the procedure, by restricting to functions with the same header: *Z←F X*, made of a certain number of instructions with the same syntax: `''⎕EA'Z←<Expr>'`.
- The number of instructions to be generated from a given genotype (a number between 0 and 255), is represented by the first integer in the genotype.
- The BNF following grammar was used:

```
<Expr>::=<Opd><Opr><Opd>
<Expr>::=<Opr><Opd>
<Expr>::=<Opd>
<Opr>::=+
<Opr>::=-
<Opr>::=×
<Opr>::=*
<Opr>::=÷
<Opr>::=○
<Opr>::=⌈
<Opr>::=⌊
<Opr>::=⍟
<Opr>::=!
<Opd>::=0
<Opd>::=1
<Opd>::=2
<Opd>::=3
<Opd>::=4
<Opd>::=5
<Opd>::=6
<Opd>::=7
<Opd>::=8
<Opd>::=9
<Opd>::=X  (three identical rules)
<Opd>::=Z  (seven identical rules)
<Opd>::=(<Expr>)  (ten identical rules)
```

Let us see what this means:

- Only monadic APL functions are generated made of a random number of similar instructions which will be executed successively, for no *go to* instructions changing the order of execution are allowed.

- Every instruction is protected by the ⎕*EA* system function, so that no errors may occur during the execution of the generated function. Grammatical evolution guarantees that syntax errors may not happen, but semantic and execution errors (such as divisions by zero) could still occur.
- The three rules for the axiom of the grammar (<*Expr*>) indicate that an expression may be either the result of a dyadic operation, the result of a monadic operation, or a single operand.
- APL primitive functions are restricted to ten: + - × * ÷ ○ ⌈ ⌊ ⍟ !. All of them may be either monadic or dyadic.
- Constants are restricted to one digit positive integers (numbers from 0 to 9).
- Variables are restricted to *X* (the function right argument) and *Z* (the function result).
- An operand may be an integer constant (with 33.3% probability), variable *X* (10% probability), variable Z (23.3% probability), or a new expression (33.3% probability). These probabilities are the automatic result of the number of rules for the <Opd> non-terminal symbol, e.g. in 10 rules out of 30 the right hand side is an integer constant, and therefore its probability is 1/3.
- Two samples of APL expressions which can be generated by means of this grammar are:

  ```
  (X+1)×(X-3)
  Z+(X*(6⌊2))
  ```

The following APL2 function implements the genetic algorithm associated to the grammatical evolution procedure described above:

```
[0]   Z←FIT EVOL OBJ;I;F;G;G1;O1;O2;⎕IO
[1]   ⎕IO←0
[2]   G←?(100+?100)⍴256 ⍝ Generates the initial genotype
[3]   I←0               ⍝ Initial counter
[4]   O1←∈OBJ TEST G    ⍝ Translates genotype-phenotype, computes fitness
[5]  L:→(O1≤FIT)/FIN    ⍝ Test for end of process
[6]   G1←G              ⍝ Prepare new generation
[7]   →(1=?2)/L0        ⍝ Apply mutation/elision with 50% probability
[8]   G1←((?⍴G1)⌽0,(¯1+⍴G1)⍴1)/G1 ⍝ This performs elision of one element
[9]   →L1
[10] L0:G1[?⍴G1]←?256   ⍝ This performs mutation of one random element
[11] L1:O2←∈OBJ TEST G1 ⍝ Compute fitness of new genotype
[12]  →(O2>O1)/L2       ⍝ Is it better than the previous one?
[13]  G←G1⋄O1←O2        ⍝ Yes, replace old genotype by new
[14] L2:→(10000≥I←I+1)/L⍝ Loop back and try again
[15] FIN:Z←G O1(⎕CR 'F')⍝ End of the process:
                        ⍝ return genotype, fitness and generated function
```

Notice that:

1. The population is made of a single genotype.
2. Function *TEST* translates the genotype into the corresponding phenotype and computes its fitness.
3. The left argument *FIT* is the minimum fitness value acceptable as a final result. The optimal fitness is assumed to be zero.
4. The right argument *OBJ* describes the target function to be generated by means of a two row matrix: the first row contains a set of values for the independent argument of the function (variable *X*), the second row the

values that the function must generate (variable *Z*). Fitness is computed by a least-square method.

5. The procedure is repeated 10000 times, or until a genotype is generated with the required minimum fitness.

6. Functions generated in this way may contain lots of *garbage code*, instructions which do nothing (dead assignments to variables which are never used). In the results shown later, all the garbage code, as well as the ⎕*EA* protection, has been eliminated manually.

Function *TEST* is shown next:

```
[0]   Z←V TEST X;N;⎕IO;E;I;P;K;Q
[1]   ⎕IO←I←0
[2]   Z←'Z←F X' 'Z←X' ⍝ Generates function header
[3]   N←↑X            ⍝ Nr. of lines to generate
[4]   X←1↓X           ⍝ Remainder of genome
[5]  L:→(N=0)/FIN     ⍝ No more lines to be generated?
[6]   E←,'E'          ⍝ Axiom for one line (expression)
[7]  L1:→(∧/~E∈'EO⍟')/L2 ⍝ End if no more non-terminal symbols in string
[8]   P←↑I⌽X          ⍝ Get next genome element
[9]   K←(3 10 30)['EO⍟'⍳E[Q←⌊/E⍳'EO⍟']] ⍝ Number of rules for this symbol
[10]  →(K=3 10 30)/L11,L12,L13 ⍝ Jump according to symbol
[11] L11:→L14,⍴E←(Q↑E),((K|P)⊃'⍟' 'O⍟' 'OO⍟'),(Q+1)↓E ⍝ Expression
[12] L12:→L14,⍴E←(Q↑E),((K|P)⊃'+-×*÷○⌈⌊⍟!'),(Q+1)↓E ⍝ Operator
[13] L13:→L14,⍴E←(Q↑E),((K|P)⊃'0123456789XXXZZZZZZZ',10⍴⊂'(E)'),(Q+1)↓E
                      ⍝ Rules for operand
[14] L14:→(5000<I←I+1)/FIN◊→L1 ⍝ Loop back (or abandon if too many loops)
[15] L2:Z←Z,⊂''''' ⎕EA ''Z←',E,'''' ⍝ Add new line to generated function
[16]  →L,N←N-1        ⍝ Loop back to generate next line
[17] FIN:⎕WA←⎕FX Z    ⍝ Generation ends. Create generated function
[18]  Z←F⍨V[0;]       ⍝ Execute function on provided arguments
[19]  Z←'1000' ⎕EA '((+/(|V[1;]-Z)*2)*0.5)×(⍴X)÷100'
                      ⍝ Compare results with target, compute fitness
```

Results

The preceding procedure was executed ten times with the following arguments and different random seeds:

```
0.1 EVOL 2 5⍴1 2 3 4 5 1 4 9 16 25
```

In nine out of the ten cases, a perfect fitness solution was obtained. In five of them, the function generated was the following:

```
[0] Z←F X
[1] Z←X×X
```

Another case gave the equivalent answer:

```
[0] Z←F X
[1] Z←(X)×(X)
```

The seventh solution was also equivalent:

```
[0] Z←F X
[1] Z←X
[2] Z←Z×Z
```

The eighth gave a different equivalent answer:

```
[0] Z←F X
[1] Z←X*2
```

The ninth was more convoluted, but also correct:

```
[0] Z←F X
[1] Z←0
[2] Z←Z+(X*(6⌊2))
```

The tenth was the most complicated, besides being only approximate:

```
[0] Z←F X
[1] Z←⌊X
[2] Z←Z×X
[3] Z←(⌊X)
[4] ''⎕EA'Z←Z○Z'
[5] Z←Z
```

In the next experiment, the following line was executed ten times with different random seeds:

```
0.1 EVOL 2 5⍴1 4 9 16 25 1 2 3 4 5
```

This is the inverse function to the former. With the provided operators, it's more difficult to find an exact solution. The program ended in seven cases with an approximate solution. Another case gave a perfect fitness answer, valid for the arguments provided, but which was not the answer that one would have been foreseen: `Z←X*÷2`. The solution found was:

```
[0] Z←F X
[1] Z←(○2)⊛X
[2] Z←○((Z))
[3] Z←⌊Z
[4] ''⎕EA'Z←Z÷0'
```

Try it! Surprisingly, it gives the correct answer.

In the future, the subset of the APL grammar may be extended to let more complicated programs be generated. Further experiments should also be performed with different target functions.

References

[Koza 1992] Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Massachusetts. 1992.

[ONei11 2001] M. O'Neill and C. Ryan: *Grammatical Evolution*, IEEE Trans. on Evolutionary Computation, Vol. 5:4, p. 349–358, 2001.

[O'Neill and Conor 2003] O'Neill, M., Conor, R.: *Grammatical Evolution, evolutionary automatic programming in an arbitrary language*, Kluwer Academic Publishers, 2003.

EXERCISES AND PROBLEMS

By Ray Polivka

Solutions to the Diagonal Sums on Spiral Matrices

The answer to the sums of the diagonals of a 1001 by 1001 spiral matrix is 669,171,001. More interesting than the answer is the number of responses and the variety of solutions. Beside solutions from APL and J, I received solutions using C++, RoboBasic, and EXCEL. Several submitters used their observations to produce mathematical solutions. Some of the mathematical solutions resulted in different equations. I'll try to summarize all the submissions.

Peter Weidner produced the recurrence relation

$$S(N+2) = S(N) + 4N^2 + 10(N+1)$$

and solved it yielding

$$S(N) = 1/6(N+1)(4N^2 - N + 9) - 3$$

or in APL, with

```
      N←1001
      ANS← ¯3+(÷6)×(N+1)×(4×N*2)+9-N
      ANS
669171001
```

Ralph Selfridge observed patterns going from the number one up each of the four semi diagonals. Going counter-clockwise starting at the top right corner, the differences between successive numbers on the diagonals were

```
4(N-1),4(N-1)-2,4(N-2) and 4(N-2)-2
```

where N is the number of items in the next square edge. Thus going from 9 to 25 N is 5 and the difference is 16. He derived the following function which summed together each of the four semi diagonals:

```
[0]   Z←LP N;⎕IO
[1]   ⎕IO←1
[2]   Z←¯3++/+\4,¯12+32×ι⌊N÷2
```