Grammatical Evolution*

Jason Brownlee
jasonb@CleverAlgorithms.com
The Clever Algorithms Project
http://www.CleverAlgorithms.com

April 3, 2010
Technical Report: CA-TR-20100403-1

Abstract

The Clever Algorithms project aims to describe a large number of Artificial Intelligence algorithms in a complete, consistent, and centralized manner, to improve their general accessibility. The project makes use of a standardized algorithm description template that uses well-defined topics that motivate the collection of specific and useful information about each algorithm described. This report describes the Grammatical Evolution algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Grammatical, Evolution`

1 Introduction

The Clever Algorithms project aims to describe a large number of algorithms from the fields of Computational Intelligence, Biologically Inspired Computation, and Metaheuristics in a complete, consistent and centralized manner [1]. The project requires all algorithms to be described using a standardized template that includes a fixed number of sections, each of which is motivated by the presentation of specific information about the technique [2]. This report describes the Grammatical Evolution algorithm using the standardized template.

2 Name

Grammatical Evolution, GE

3 Taxonomy

Grammatical Evolution is a Global Optimization technique and an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. It may also be considered an algorithm for Automatic Programming. Grammatical Evolution is related to other Evolutionary Algorithms for evolving programs such as Genetic Programming, as well as the classical Genetic Algorithm that uses binary strings.

4 Inspiration

The Grammatical Evolution algorithm is inspired by the biological process used for generating a protein from genetic material as well as the broader genetic evolutionary process. The genome is comprised of DNA as a string of building blocks that are transcribed to RNA. RNA codons are in turn translated into sequences of amino acids and used in the protein. The resulting protein in its environment is the phenotype.

5 Metaphor

The phenotype is a computer program that is created from a binary string-based genome. The genome is decoded into a sequence of integers that are in turn mapped onto pre-defined rules that makeup the program. The mapping from genotype to the phenotype is many-to-many process that uses a wrapping feature. This is like the biological process observed in many bacteria, viruses, and mitochondria, where the same genetic material is used in the expression of different genes. The mapping adds robustness to the process both in the ability to adopt structure-agnostic genetic operators used during the evolutionary process on the sub-symbolic representation and the transcription of well-formed executable programs from the representation.

6 Strategy

The objective of Grammatical Evolution is to adapt an executable program to a problem specific objective function. This is achieved through an iterative process with surrogates of evolutionary mechanisms such as descent with variation, genetic mutation and recombination, and genetic transcription and gene expression. A population of programs are evolved in a sub-symbolic form as variable length binary strings and mapped to a symbolic and well-structured form as a context free grammar for execution.

7 Procedure

A grammar is defined in Backus Normal Form (BNF), which is a context free grammar expressed as a series of production rules comprised of terminals and non-terminals. A variable-length binary string representation is used for the optimization process. Bits are read from the a candidate solutions genome in blocks of 8 and decoded to an integer (in the range between 0 and 2^{8-1}). If the end of the binary string is reached when reading integers, the reading process loops back to the start of the string, effectively creating a circular genome. The integers are mapped to expressions from the BNF until a complete syntactically correct expression is formed. This may not use a solutions entire genome, or use the decoded genome more than once given it's circular nature. Algorithm 1 provides a pseudo-code listing of the Grammatical Evolution algorithm for minimizing a cost function.

8 Heuristics

- Grammatical Evolution was designed to optimize programs (such as mathematical equations) to specific cost functions.
- Classical genetic operators used by the Genetic Algorithm may be used in the Grammatical Evolution algorithm, such as point mutations and one-point crossover.

Algorithm 1: Pseudo Code for the Grammatical Evolution algorithm.

```
Input: Grammar, Codon_numbits Population_size, P_crossover, P_mutation, P_delete, P_duplicate
Output: S_best
Population ← InitializePopulation(Population_size, Codon_numbits);
foreach S_i ∈ Population do
    Si_integers ← Decode(Si_bitstring, Codon_numbits);
    Si_program ← Map(Si_integers, Grammar);
    Si_cost ← Execute(Si_program);
end
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    Parents ← SelectParents(Population, Population_size);
    Children ← 0;
    foreach Parent_i, Parent_j ∈ Parents do
        S_i ← Crossover(Parent_i, Parent_j, P_crossover);
        Si_bitstring ← CodonDeletion(Si_bitstring, P_delete);
        Si_bitstring ← CodonDuplication(Si_bitstring, P_duplicate);
        Si_bitstring ← Mutate(Si_bitstring, P_mutation);
        Children ← S_i;
    end
    foreach S_i ∈ Children do
        Si_integers ← Decode(Si_bitstring, Codon_numbits);
        Si_program ← Map(Si_integers, Grammar);
        Si_cost ← Execute(Si_program);
    end
    S_best ← GetBestSolution(Children);
    Population ← Replace(Population, Children);
end
return S_best;
```

- Codon's (groups of bits mapped to an integer) are commonly fixed at 8-bits, proving a range of integers $\in [0, 2^{8-1}]$ that may be scaled to the range of rules using a modulo function.
- Additional genetic operators may be used with variable-length representations such as codon duplication (add to the end) and deletion.

9 Code Listing

Listing 1 provides an example of the Grammatical Evolution algorithm implemented in the Ruby Programming Language based on the version described by O'Neill and Ryan [8]. The demonstration problem is an instance of symbolic regression $f(x) = x^4 + x^3 + x^2 + x$, where $x \in [-1, 1]$. The grammar used in this problem is:

- Non-terminals: $N = \{expr, op, pre\_op\}$
- Terminals: $T = \{sin, cos, exp, log, +, -, /, *, x, 1.0\}$
- Expression (program): $S =< expr >$

The production rules for the grammar in BNF are:

- $< expr > ::= < expr >< op >< expr >$, $(< expr >< op >< expr >)$, $< pre\_op > (< expr >)$, $< var >$
- $< op > ::= +, -, \div, \times$
- $< pre\_op > ::=$ Sin, Cos, Exp, Log
- $< var > ::=$ x, 1.0

The algorithm uses point mutation and a codon-respecting one-point crossover operator. Binary tournament selection is used to determine the parent population's contribution to the subsequent generation. Binary strings are decoded to integers using the Binary Coded Decimal method. Candidate solutions are then mapped directly into executable ruby code and executed. A given candidate solution is evaluated by comparing its output against the target function and taking the sum of the absolute errors over a number of trials. The probabilities of point mutation, codon deletion, and codon duplication are hard coded as relative probabilities to each solution, although should be parameters of the algorithm. In this case they are heuristically defined as $\frac{1.0}{L}$, $\frac{0.5}{NC}$ and $\frac{1.0}{NC}$ respectively, where L is the total number of bits, and NC is the number of codons in a given candidate solution.

```
def binary_tournament(population)
  s1, s2 = population[rand(population.size)], population[rand(population.size)]
  return (s1[:fitness] > s2[:fitness]) ? s1 : s2
end

def point_mutation(bitstring)
  rate = 1.0/bitstring.to_f
  child = ""
  bitstring.size.times do |i|
    bit = bitstring[i]
    child << ((rand()<rate) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def one_point_crossover(parent1, parent2, p_crossover, codon_bits)
  return ""+parent1[:bitstring] if rand()>=p_crossover
  cut = rand([parent1.length, parent2.length].min/codon_bits)
  cut *= codon_bits
  p2length = parent2[:bitstring].length
  return parent1[:bitstring][0...cut]+parent2[:bitstring][cut...p2length]
end

def codon_duplication(bitstring, codon_bits)
  codons = bitstring.length/codon_bits
  return bitstring if rand() >= 1.0/codons.to_f
  return bitstring + bitstring[rand(codons)*codon_bits, codon_bits]
end

def codon_deletion(bitstring, codon_bits)
  codons = bitstring.length/codon_bits
  return bitstring if rand() >= 0.5/codons.to_f
  off = rand(codons)*codon_bits
  return bitstring[0...off] + bitstring[off+codon_bits...bitstring.length]
end

def reproduce(selected, population_size, p_crossover, codon_bits)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:bitstring] = one_point_crossover(p1, p2, p_crossover, codon_bits)
```

```
    child[:bitstring] = codon_deletion(child[:bitstring], codon_bits)
    child[:bitstring] = codon_duplication(child[:bitstring], codon_bits)
    child[:bitstring] = point_mutation(child[:bitstring])
    children << child
  end
  return children
end

def random_bitstring(num_bits)
  return (0...num_bits).inject(""){|s,i| s<<((rand<0.5) ? "1" : "0")}
end

def decode_integers(bitstring, codon_bits)
  ints = []
  (bitstring.length/codon_bits).times do |off|
    codon = bitstring[off*codon_bits, codon_bits]
    sum, i = 0, 0
    codon.each_char {|x| sum+=((x=='1') ? 1 : 0) * (2 ** i);i+=1}
    ints << sum
  end
  return ints
end

def map(grammar, integers, max_depth)
  done, offset, depth = false, 0, 0
  symbolic_string = grammar["S"]
  begin
    done = true
    grammar.keys.each do |key|
      symbolic_string = symbolic_string.gsub(key) do |k|
        done = false
        set = (k=="EXP" and depth>=max_depth-1) ? grammar["VAR"] : grammar[k]
        integer = integers[offset].modulo(set.length)
        offset = (offset==integers.length-1) ? 0 : offset+1
        set[integer]
      end
    end
    depth += 1
  end until done
  return symbolic_string
end

def target_function(x)
  x**4.0 + x**3.0 + x**2.0 + x
end

def cost(program, bounds)
  errors = 0.0
  10.times do
    x = bounds[0] + ((bounds[1] - bounds[0]) * rand())
    expression = program.gsub("INPUT", x.to_s)
    target = target_function(x)
    begin score = eval(expression) rescue score = 0.0/0.0 end
    errors += (((score.nan? or score.infinite?) ? 0.0 : score) - target).abs
  end
  return errors
end

def evaluate(candidate, codon_bits, grammar, max_depth, bounds)
  candidate[:integers] = decode_integers(candidate[:bitstring], codon_bits)
  candidate[:program] = map(grammar, candidate[:integers], max_depth)
  candidate[:fitness] = cost(candidate[:program], bounds)
end
```

```ruby

def search(generations, pop_size, codon_bits, initial_bits, p_crossover, grammar, max_depth, bounds)
  pop = Array.new(pop_size) {|i| {:bitstring=>random_bitstring(initial_bits)}}
  pop.each{|c| evaluate(c,codon_bits, grammar, max_depth, bounds)}
  gen, best = 0, pop.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  generations.times do |gen|
    selected = Array.new(pop_size){|i| binary_tournament(pop)}
    children = reproduce(selected, pop_size, p_crossover,codon_bits)
    children.each{|c| evaluate(c,codon_bits, grammar, max_depth, bounds)}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    pop = children
    puts " > gen=#{gen}, f=#{best[:fitness]}, codons=#{best[:bitstring].length/codon_bits}, s=#{best[:bitstring]}"
  end
  return best
end

grammar = {"S"=>"EXP",
  "EXP"=>[" EXP BINARY EXP ", " (EXP BINARY EXP) ", " UNIARY(EXP) ", " VAR "],
  "BINARY"=>["+", "-", "/", "*" ],
  "UNIARY"=>["Math.sin", "Math.cos", "Math.exp", "Math.log"],
  "VAR"=>["INPUT", "1.0"]}
max_depth = 7
bounds = [-1, +1]
generations = 100
pop_size = 100
codon_bits = 8
initial_bits = 10*codon_bits
p_crossover = 0.30

best = search(generations, pop_size, codon_bits, initial_bits, p_crossover, grammar, max_depth, bounds)
puts "done! Solution: f=#{best[:fitness]}, s=#{best[:program]}"
```

Listing 1: Grammatical Evolution algorithm in the Ruby Programming Language

10 References

10.1 Primary Sources

Grammatical Evolution was proposed by Ryan, Collins and O'Neill in a seminal conference paper that applied the approach to a symbolic regression problem [9]. The approach was born out of the desire for syntax preservation while evolving programs using the Genetic Programming algorithm. This seminal work was followed by application papers for a symbolic integration problem [4, 5] and solving trigonometric identities [10].

10.2 Learn More

O'Neill and Ryan provide a high-level introduction to Grammatical Evolution and early demonstration applications [6]. The same authors provide a through introduction to the technique and overview of the state of the field [8]. O'Neill and Ryan present a seminal reference for Grammatical Evolution in their book [7]. A second more recent book considers extensions to the approach improving its capability on dynamic problems [3].

11 Conclusions

This report described the Grammatical Evolution algorithm as a automatic programming technique that uses context free grammars as a representation, and ensures that structurally correct programs are always created.

12 Contribute

Found a typo in the content or a bug in the source code? Are you an expert in this technique and know some facts that could improve the algorithm description for all? Do you want to get that warm feeling from contributing to an open source project? Do you want to see your name as an acknowledgment in print?

Two pillars of this effort are i) that the best domain experts are people outside of the project, and ii) that this work is wrong by default. Please help to make this work less wrong by emailing the author 'Jason Brownlee' at jasonb@CleverAlgorithms.com or visit the project website at http://www.CleverAlgorithms.com.

References

[1] Jason Brownlee. The clever algorithms project: Overview. Technical Report CA-TR-20100105-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[2] Jason Brownlee. A template for standardized algorithm descriptions. Technical Report CA-TR-20100107-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

[3] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*. Springer, 2009.

[4] M. O'Neill and C. Ryan. Grammatical evolution: A steady state approach. In *Proceedings of the Second International Workshop on Frontiers in Evolutionary Algorithms*, pages 419–423, 1998.

[5] M. O'Neill and C. Ryan. Grammatical evolution: A steady state approach. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, 1998.

[6] M. O'Neill and C. Ryan. Under the hood of grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 1999.

[7] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Springer, 2003.

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

[9] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming*, 1998.

[10] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Solving trigonometric identities. In *Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimisation Problems, Fuzzy Logic, Neural Networks, Rough Sets.*, pages 111–119, 1998.