Gene Expression Programming*

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

April 4, 2010
Technical Report: CA-TR-20100404-1

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 Gene Expression Programming algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Gene, Expression, Programming`

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 Gene Expression Programming algorithm using the standardized template.

2 Name

Gene Expression Programming, GEP

3 Taxonomy

Gene Expression Programming is a Global Optimization algorithm and an Automatic Programming technique, and it is an instance of an Evolutionary Algorithm from the field of Evolutionary Computation. It is a sibling of other Evolutionary Algorithms such as a the Genetic Algorithm as well as other Evolutionary Automatic Programming techniques such as Genetic Programming and Grammatical Evolution.

4 Inspiration

Gene Expression Programming is inspired by the replication and expression of the DNA molecule, specifically at the gene level. The expression of a gene involves the transcription of its DNA to RNA which in turn forms amino acids that make up proteins in the phenotype of an organism. The DNA building blocks are subjected to mechanisms of variation (mutations such as coping errors) as well as recombination during sexual reproduction.

5 Metaphor

Gene Expression Programming uses a linear genome as the basis for genetic operators such as mutation, recombination, inversion, and transposition. The genome is comprised of chromosomes and each chromosome is comprised of genes that are translated into an expression tree to solve a given problem. The robust gene definition means that genetic operators can be applied to the sub-symbolic representation without concern for the structure of the resultant gene expression, providing separation of genotype and phenotype.

6 Strategy

The objective of the Gene Expression Programming algorithm is to improve the adaptive fit of an expressed program in the context of a problem specific cost function. This is achieved through the use of an evolutionary process that operates on a sub-symbolic representation of candidate solutions using surrogates for the processes (descent with modification) and mechanisms (genetic recombination, mutation, inversion, transposition, and gene expression) of evolution.

7 Procedure

A candidate solution is represented as a linear string of symbols called Karva notation or a K-expression, where each symbol maps to a function or terminal node. The linear representation is mapped to an expression tree in a breadth-first manner. A K-expression is fixed length and is comprised of one or more sub-expressions (genes), which are also defined with a fixed length. A gene is comprised of two sections, a head which may contain any function or terminal symbols, and a tail section that may only contain terminal symbols. Each gene will always translate to a syntactically correct expression tree, where the tail portion of the gene provides a genetic buffer which ensures closure of the genes expression.

Algorithm 1 provides a pseudo-code listing of the Gene Expression Programming algorithm for minimizing a cost function.

8 Heuristics

- The length of a chromosome is defined by the number of genes, where a gene length is defined by $h + t$. The h is a user defined parameter (such as 10), where as t is defined as $t = h(n-1)+1$. The n represents the maximum arity of functional nodes in the expression (such as 2 if the arithmetic functions $\times, \div, -, +$ are used).
- The mutation operate substituted expressions along the genome, although must respect the gene rules such that function and terminal nodes are mutated in the head of genes, whereas only terminal nodes are substituted in the tail of genes.

Algorithm 1: Pseudo Code for the Gene Expression Programming algorithm.

Input: Grammar, $Population_{size}$, $Head_{length}$, $Tail_{length}$, $P_{crossover}$, $P_{mutation}$
Output: S_{best}

```
Population ← InitializePopulation(Population_size, Grammar, Head_length, Tail_length);
foreach S_i ∈ Population do
    Si_program ← DecodeBreadthFirst(Si_genome, Grammar);
    Si_cost ← Execute(Si_program);
end
S_best ← GetBestSolution(Population);
while ¬StopCondition() do
    Parents ← SelectParents(Population, Population_size);
    Children ← 0;
    foreach Parent_1, Parent_2 ∈ Parents do
        Si_genome ← Crossover(Parent_1, Parent_2, P_crossover);
        Si_genome ← Mutate(Si_genome, P_mutation);
        Children ← S_i;
    end
    foreach S_i ∈ Population do
        Si_program ← DecodeBreadthFirst(Si_genome, Grammar);
        Si_cost ← Execute(Si_program);
    end
    S_best ← GetBestSolution(Children);
    Population ← Replace(Population, Children);
end
return S_best;
```

- Crossover occurs between two selected parents from the population and can occur based on a one-point cross, two point cross, or finally a gene-based approach were genes are selected from the parents with uniform probability.
- An inversion operator may be used with a low probability that reverses a small sequence of symbols (1-3) within a section of a gene (tail or head).
- A transposition operator may be used that has a number of different modes, including: duplicate a small sequences (1-3) from somewhere on a gene to the head, small sequences on a gene to the root of the gene, and moving of entire genes on the chromosome. In the case of intra-gene transpositions, the sequence in the head of the gene is moved down to accommodate the copied sequence and the length of the head is truncated to maintain consistent gene sizes.
- A '?' is included in the terminal set that represents a numeric constant from an array that are evolved on the end of the genome. The constants are read from the end of the genome and are substituted for '?' as the expression tree is created (in breadth first order). Finally the numeric constants are used as array indices in yet another chromosome of numerical values which are substituted into the expression tree.
- Mutation is low (such as $\frac{1}{L}$), selection can be any of the classical approaches (such as roulette wheel or tournament), and crossover rates are typically high (0.7 of offspring)
- Use multiple sub-expressions linked together on hard problems when one gene does not get much progress. The sub-expressions are linked using link expressions which are function

nodes that are either statically defined (such as a conjunction) or evolved on the genome with the genes.

9 Code Listing

Listing 1 provides an example of the Gene Expression Programming algorithm implemented in the Ruby Programming Language based on the seminal version proposed by Ferreira [3]. 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: Functions: $F = \{+, -, \div, \times, \}$ and Terminals: $T = \{x\}$. The algorithm uses binary tournament selection, uniform crossover and point mutations. The K-expression is decoded to an expression tree in a breadth-first manner, which is then parsed depth first as an ruby expression string for display and direct evaluation.

```ruby
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(grammar, genome, p_mutation, head_length)
  child, i = "", 0
  genome.each_char do |v|
    if rand()<p_mutation
      if (i<head_length)
        child << grammar["FUNC"][rand(grammar["FUNC"].length)]
      else
        child << grammar["TERM"][rand(grammar["TERM"].length)]
      end
    else
      child << v
    end
    i += 1
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1 if rand()>=p_crossover
  child = ""
  parent1.length.times do |i|
    child << ((rand()<0.5) ? parent1[i] : parent2[i])
  end
  return child
end

def reproduce(grammar, selected, pop_size, p_crossover, p_mutation, head_length)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:genome] = uniform_crossover(p1[:genome], p2[:genome], p_crossover)
    child[:genome] = point_mutation(grammar, child[:genome], p_mutation, head_length)
    children << child
  end
  return children
end

def random_genome(grammar, head_length, tail_length)
  s = ""
  head_length.times { s<<grammar["FUNC"][rand(grammar["FUNC"].length)]}
  tail_length.times { s<<grammar["TERM"][rand(grammar["TERM"].length)]}
  return s
```

```ruby
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("x", 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 breadth_first_mapping(genome, grammar)
  off, queue = 0, Array.new
  root = {}
  root[:node] = genome[off].chr;off+=1
  queue.push(root)
  while !queue.empty? do
    current = queue.shift
    if grammar["FUNC"].include?(current[:node])
      current[:left] = {}
      current[:left][:node] = genome[off].chr;off+=1
      queue.push(current[:left])
      current[:right] = {}
      current[:right][:node] = genome[off].chr;off+=1
      queue.push(current[:right])
    end
  end
  return root
end

def tree_to_string(exp)
  return exp[:node] if (exp[:left].nil? and exp[:right].nil?)
  left = tree_to_string(exp[:left])
  right = tree_to_string(exp[:right])
  return "(#{left} #{exp[:node]} #{right})"
end

def evaluate(candidate, grammar, bounds)
  candidate[:expression] = breadth_first_mapping(candidate[:genome], grammar)
  candidate[:program] = tree_to_string(candidate[:expression])
  candidate[:fitness] = cost(candidate[:program], bounds)
end

def search(grammar, bounds, head_length, tail_length, generations, pop_size, p_crossover,
    p_mutation)
  pop = Array.new(pop_size) do
    {:genome=>random_genome(grammar, head_length, tail_length)}
  end
  pop.each{|c| evaluate(c, grammar, bounds)}
  gen, best = 0, pop.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  generations.times do |gen|
    selected = Array.new(pop){|i| binary_tournament(pop)}
    children = reproduce(grammar, selected, pop_size, p_crossover, p_mutation, head_length)
    children.each{|c| evaluate(c, grammar, bounds)}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
```

```
    pop = children
    gen += 1
    puts " > gen=#{gen}, f=#{best[:fitness]}, g=#{best[:genome]}"
  end
  return best
end

grammar = {"FUNC"=>["+","-","*","/"], "TERM"=>["x"]}
bounds = [-1, 1]
head_length = 24
tail_length = head_length * (2-1) + 1
generations = 150
pop_size = 100
p_crossover = 0.70
p_mutation = 2.0/(head_length+tail_length).to_f

best = search(grammar, bounds, head_length, tail_length, generations, pop_size, p_crossover,
    p_mutation)
puts "done! Solution: f=#{best[:fitness]}, g=#{best[:genome]}, b=#{best[:program]}"
```

Listing 1: Gene Expression Programming algorithm in the Ruby Programming Language

10 References

10.1 Primary Sources

The Gene Expression Programming algorithm was proposed by Ferreira in a paper that detailed the approach, provided a careful walkthrough of the process and operators and demonstrated the the algorithm on a number of benchmark problem instances such as symbolic regression [3].

10.2 Learn More

Ferreira provided an early and detailed introduction and overview of the approach as book chapter, providing a step-by-step walkthrough of the procedure and sample applications [4]. A similar more contemporary and detailed introduction is provided in a second book chapter [5]. Ferreira published a book on the approach in 2002 covering background, the algorithm, and demonstration applications which is now in its second edition [6].

11 Conclusions

This report described the Gene Expression Programming algorithm as an evolutionary automatic programming technique.

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 (somewhat) 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] C. Ferreira. Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129, 2001.

[4] C. Ferreira. *Soft Computing and Industry: Recent Applications*, chapter Gene Expression Programming in Problem Solving, pages 635–654. Springer-Verlag, 2002.

[5] C. Ferreira. *Recent Developments in Biologically Inspired Computing*, chapter Gene Expression Programming and the Evolution of computer programs, pages 82–103. Idea Group Publishing, 2005.

[6] Candida Ferreira. *Gene expression programming: mathematical modeling by an artificial intelligence*. Springer-Verlag, second edition, 2006.