Genetic Programming*

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

March 8, 2010
Technical Report: CA-TR-20100308-1

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 Genetic Programming algorithm using the standardized template.

Keywords: `Clever, Algorithms, Description, Optimization, Genetic, 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 [3]. 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 [4]. This report describes the Genetic Programming algorithm using the standardized template.

2 Name

Genetic Programming, GP

3 Taxonomy

The Genetic Programming algorithm is an example of a Evolutionary Algorithm (EA) and belongs to the field of Evolutionary Computation (EC) and more broadly Computational Intelligence and Biologically Inspired Computation. The Genetic Programming algorithm is a sibling to other Evolutionary Algorithms such as the Genetic Algorithm, Evolution Strategies, Evolutionary Programming, and Learning Classifier Systems. Technically, the Genetic Programming algorithm is an extension of the Genetic Algorithm. The Genetic Algorithm is a parent to a host of variations and extensions.

4 Inspiration

The Genetic Algorithm is inspired by population genetics (including heredity and gene frequencies), and evolution at the population level, as well as the Mendelian understanding of the structure (such as chromosomes, genes, alleles) and mechanisms (such as recombination and mutation). This is the so-called new or modern synthesis of evolutionary biology.

5 Metaphor

Individuals of a population contribute their genetic material (called the genotype) proportional to their suitability of their expressed genome (called their phenotype) to their environment. The next generation is created through a process of mating that involves genetic operators such as recombination of two individuals genomes in the population and the introduction of random copying errors (called mutation). This iterative process may result in an improved adaptive-fit between the phenotypes of individuals in a population and the environment.

Programs may be evolved and used in a secondary adaptive process, where an assessment of candidates at the end of the secondary adaptive process is used for differential reproductive success in the first evolutionary process. This system may be understood as the inter-dependencies experienced in evolutionary development where evolution operates upon an embryo that in turn develops into an individual in an environment that eventually may reproduce.

6 Strategy

The objective of the Genetic Programming algorithm is to use induction to devise a computer program. This is achieved by using evolutionary operators on candidate programs with a tree structure to improve the adaptive fit between the population of candidate programs and an objective function. An assessment of a candidate solution involves its execution.

7 Procedure

Algorithm 1 provides a pseudo-code listing of the Genetic Programming algorithm for minimizing a cost function, based on Koza and Poli's tutorial [11].

The Genetic Program uses LISP-like symbolic expressions called S-expressions that represent the graph of a program with function nodes and terminal nodes. While the algorithm is running, the programs are treated like data, and when they are evaluated they are executed. The traversal of a program graph is always depth first, and functions must always return a value.

8 Heuristics

- The Genetic Programming algorithm was designed for inductive automatic programming and is well suited to symbolic regression, controller design, and machine learning tasks under the broader name of function approximation.
- Traditionally symbolic expressions are evolved and evaluated in a virtual machine, although the approach has been applied with real programming languages.
- The evaluation (fitness assignment) of a candidate solution typically takes the structure of the program into account, rewarding parsimony.
- The selection process should be balanced between random selection and greedy selection to bias the search towards fitter candidate solutions (exploitation), whilst promoting useful diversity into the population (exploration).

Algorithm 1: Pseudo Code for the Genetic Programming algorithm.

Input: $Population_{size}$, $nodes_{func}$, $nodes_{term}$, $P_{crossover}$, $P_{mutation}$, $P_{reproduction}$, $P_{alteration}$
Output: S_{best}

Population $\leftarrow$ `InitializePopulation(`$Population_{size}$, $nodes_{func}$, $nodes_{term}$`)`;
`EvaluatePopulation(`Population`)`;
$S_{best} \leftarrow$ `GetBestSolution(`Population`)`;
while $\neg$`StopCondition()` **do**
    Children $\leftarrow 0$;
    while $\neg$`StopCondition(Size(`Children`)` $<$ $Population_{size}$`)` **do**
        Operator $\leftarrow$ `SelectGeneticOperator(`$P_{crossover}$, $P_{mutation}$, $P_{reproduction}$, $P_{alteration}$`)`;
        if Operator $\equiv$ CrossoverOperator **then**
            $Parent_1$, $Parent_2$ $\leftarrow$ `SelectParents(`Population, $Population_{size}$`)`;
            $Child_1$, $Child_2$ $\leftarrow$ `Crossover(`$Parent_1$, $Parent_2$`)`;
            Children $\leftarrow$ $Child_1$;
            Children $\leftarrow$ $Child_2$;
        end
        else if Operator $\equiv$ MutationOperator **then**
            $Parent_1$ $\leftarrow$ `SelectParents(`Population, $Population_{size}$`)`;
            $Child_1$ $\leftarrow$ `Mutate(`$Parent_1$`)`;
            Children $\leftarrow$ $Child_1$;
        end
        else if Operator $\equiv$ ReproductionOperator **then**
            $Parent_1$ $\leftarrow$ `SelectParents(`Population, $Population_{size}$`)`;
            $Child_1$ $\leftarrow$ `Reproduce(`$Parent_1$`)`;
            Children $\leftarrow$ $Child_1$;
        end
        else if Operator $\equiv$ AlterationOperator **then**
            $Parent_1$ $\leftarrow$ `SelectParents(`Population, $Population_{size}$`)`;
            $Child_1$ $\leftarrow$ `AlterArchitecture(`$Parent_1$`)`;
            Children $\leftarrow$ $Child_1$;
        end
    end
    `EvaluatePopulation(`Children`)`;
    $S_{best} \leftarrow$ `GetBestSolution(`Children, S_{best}`)`;
    Population $\leftarrow$ `Replace(`Population, Children`)`;
end
return S_{best};

- A program may respond to zero or more input values and may produce one or more outputs.
- All functions used in the function node set must return a usable result. For example, the division function must return a value (such as zero or one) when a division by zero occurs.
- All genetic operations ensure (or should ensure) that syntactically valid and executable programs are produced as a result of their application.
- The Genetic Programming algorithm is commonly configured with a high-probability of crossover ($\geq 90\%$) and a low-probability of mutation ($\leq 1\%$). Other operators such as reproduction and architecture alterations are used with moderate-level probabilities and fill in the probabilistic gap.
- Architecture altering operations are not limited to the duplication and deletion of sub-structures of a given program.
- The crossover genetic operator in the algorithm is commonly configured to select a function as a the cross-point with a high-probability ($\geq 90\%$) and low-probability of selecting a terminal as a cross-point ($\leq 10\%$).
- The function set may also include control structures such as conditional statements and loop constructs.
- The Genetic Programing algorithm can be realized as a stack-based virtual machine as opposed to a call graph [14].
- The Genetic Programming algorithm can make use of Automatically Defined Functions (ADFs) that are sub-graphs and are promoted to the status of functions for reuse and are co-evolved with the programs.
- The genetic operators employed during reproduction in the algorithm may be considered transformation programs for candidate solutions and may themselves be co-evolved in the algorithm [1].

9 Code Listing

Listing 1 provides an example of the Genetic Programming algorithm implemented in the Ruby Programming Language based on Koza and Poli's tutorial [11].

The demonstration problem is an instance of a symbolic regression, where a function must be devised to match a set of observations. In this case the target function is a quadratic polynomial $x^2 + x + 1$ where $x \in [-1, 1]$. The observations are generated directly from the target function without noise for the purposes of this example. In practical problems, if one knew and had access to the target function then the genetic program would not be required.

The algorithm is configured to search for a program with the function set $\{+, -, \times, \div\}$ and the terminal set $\{X, R\}$, where X is the input value, and R is a static random variable generated for a program $X \in [-5, 5]$. A division by zero returns a value of one. The fitness of a candidate solution is calculated by evaluating the program on range of random input values and calculating the Root Mean Squared Error (RMSE). The algorithm is configured with a 90% probability of crossover, 9% probability of reproduction (copying), and a 2% probability of mutation. For brevity, the algorithm does not implement the architecture altering genetic operation and does not bias crossover points towards functions over terminals.

```ruby
def random_num(min, max)
  return min + (max-min)*rand()
end

def print_program(node)
  return node if !node.kind_of? Array
  return "(#{node[0]}, #{print_program(node[1])}, #{print_program(node[2])})"
end

def eval_program(node, map)
  if !node.kind_of? Array
    return map[node].to_f if !map[node].nil?
    return node.to_f
  end
  arg1, arg2 = eval_program(node[1], map), eval_program(node[2], map)
  return 0 if node[0] === :/ and arg2 == 0.0
  return arg1.__send__(node[0], arg2)
end

def generate_random_program(max, funcs, terms, depth=0)
  if depth==max-1 or (depth>1 and rand()<0.1)
    t = terms[rand(terms.length)]
    return ((t=='R') ? random_num(-5.0, +5.0) : t)
  end
  depth += 1
  arg1 = generate_random_program(max, funcs, terms, depth)
  arg2 = generate_random_program(max, funcs, terms, depth)
  return [funcs[rand(funcs.length)], arg1, arg2]
end

def count_nodes(node)
  return 1 if !node.kind_of? Array
  a1 = count_nodes(node[1])
  a2 = count_nodes(node[2])
  return a1+a2+1
end

def target_function(input)
  return input**2 + input + 1
end

def fitness(program, num_trials)
  sum_error = 0.0
  num_trials.times do |i|
    input = random_num(-1.0, 1.0)
    error = eval_program(program, {'X'=>input}) - target_function(input)
    sum_error += error**2.0
  end
  return Math::sqrt(sum_error/num_trials.to_f)
end

def tournament_selection(population, num_bouts)
  best = population[rand(population.size)]
  (num_bouts-1).times do |i|
    candidate = population[rand(population.size)]
    best = candidate if candidate[:fitness] < best[:fitness]
  end
  return best
end

def replace_node(node, replacement, node_num, current_node=0)
  return replacement,(current_node+1) if current_node == node_num
  current_node += 1
```

```
  return node,current_node if !node.kind_of? Array
  a1, current_node = replace_node(node[1], replacement, node_num, current_node)
  a2, current_node = replace_node(node[2], replacement, node_num, current_node)
  return [node[0], a1, a2], current_node
end

def copy_program(node)
  return node if !node.kind_of? Array
  return [node[0], copy_program(node[1]), copy_program(node[2])]
end

def get_node(node, node_num, current_node=0)
  return node,(current_node+1) if current_node == node_num
  current_node += 1
  return nil,current_node if !node.kind_of? Array
  a1, current_node = get_node(node[1], node_num, current_node)
  return a1,current_node if !a1.nil?
  a2, current_node = get_node(node[2], node_num, current_node)
  return a2,current_node if !a2.nil?
  return nil,current_node
end

def prune(node, max_depth, terms, depth=0)
  if depth >= max_depth-1
    t = terms[rand(terms.length)]
    return ((t=='R') ? random_num(-5.0, +5.0) : t)
  end
  depth += 1
  return node if !node.kind_of? Array
  a1 = prune(node[1], max_depth, terms, depth)
  a2 = prune(node[2], max_depth, terms, depth)
  return [node[0], a1, a2]
end

def crossover(parent1, parent2, max_depth, terms)
  pt1, pt2 = rand(count_nodes(parent1)-2)+1, rand(count_nodes(parent2)-2)+1
  tree1, c1 = get_node(parent1, pt1)
  tree2, c2 = get_node(parent2, pt2)
  child1, c1 = replace_node(parent1, copy_program(tree2), pt1)
  child1 = prune(child1, max_depth, terms)
  child2, c2 = replace_node(parent2, copy_program(tree1), pt2)
  child2 = prune(child2, max_depth, terms)
  return child1, child2
end

def mutation(parent, max_depth, functions, terms)
  random_tree = generate_random_program(max_depth/2, functions, terms)
  point = rand(count_nodes(parent))
  child, count = replace_node(parent, random_tree, point)
  child = prune(child, max_depth, terms)
  return child
end

def search(max_generations, population_size, max_depth, num_trials, num_bouts, p_reproduction,
    p_crossover, p_mutation, functions, terminals)
  population = Array.new(population_size) do |i|
    {:program=>generate_random_program(max_depth, functions, terminals)}
  end
  population.each{|c| c[:fitness] = fitness(c[:program], num_trials)}
  best = population.sort{|x,y| x[:fitness] <=> y[:fitness]}.first
  max_generations.times do |gen|
    children = []
    while children.length < population_size
```

```
      operation = rand()
      parent = tournament_selection(population, num_bouts)
      child = {}
      if operation < p_reproduction
        child[:program] = copy_program(parent[:program])
      elsif operation < p_reproduction+p_crossover
        p2 = tournament_selection(population, num_bouts)
        c2 = {}
        child[:program], c2[:program] = crossover(parent[:program], p2[:program], max_depth,
            terminals)
        children << c2
      elsif operation < p_reproduction+p_crossover+p_mutation
        child[:program] = mutation(parent[:program], max_depth, functions, terminals)
      end
      children << child if children.length < population_size
    end
    children.each{|c| c[:fitness] = fitness(c[:program], num_trials)}
    population = children
    population.sort!{|x,y| x[:fitness] <=> y[:fitness]}
    best = population.first if population.first[:fitness] <= best[:fitness]
    puts " > gen #{gen}, fitness=#{best[:fitness]}"
    break if best[:fitness] == 0
  end
  return best
end

max_generations = 100
max_depth = 7
population_size = 100
num_trials = 15
num_bouts = 5
p_reproduction = 0.08
p_crossover = 0.90
p_mutation = 0.02
terminals = ['X', 'R']
functions = [:+, :-, :*, :/]

best = search(max_generations, population_size, max_depth, num_trials, num_bouts,
    p_reproduction, p_crossover, p_mutation, functions, terminals)
puts "done! Solution: f=#{best[:fitness]}, s=#{print_program(best[:program])}"
```

Listing 1: Genetic Programming algorithm in the Ruby Programming Language

10 References

10.1 Primary Sources

An early work by Cramer involved the study of a Genetic Algorithm using an expression tree structure for representing computer programs for primitive mathematical operations [5]. Koza is credited with the development of the field of Genetic Programming. An early paper by Koza referred to his hierarchical genetic algorithms as an extension to the simple genetic algorithm that use symbolic expressions (S-expressions) as a representation and were applied to a range of induction-style problems [6]. The seminal reference for the field is Koza's 1992 book on Genetic Programming [7].

10.2 Learn More

The field of Genetic Programming is vast, including many books, dedicated conferences and uncounted thousands of publications. Koza is generally credited with the development and popularizing of the field, publishing a large number of books and papers himself. Koza provides a practical introduction to the field as a tutorial [11], and provides recent overview of the broader field and usage of the technique [12].

In addition his the seminal 1992 book, Koza has released three more volumes in the series including volume II on automatically defined functions (ADFs) [8], volume III that considered the Genetic Programming Problem Solver (GPPS) for automatically defining the function set and program structure for a given problem [9], and volume IV that focuses on the human competitive results the technique is able to achieve in a routine manner [10]. All books are rich with targeted and practical demonstration problem instances.

Some additional excellent books include Banzhaf, et al's introduction to the field [2], Langdon and Poli's detailed look at the technique [13], and Poli, Langdon, and McPhee's contemporary and practical field guide to Genetic Programming [15].

11 Conclusions

This report described the Genetic Programming algorithm as an extension of the Genetic Algorithm for use as a functionally capable technique for inductive automatic programming.

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] Peter J. Angeline. Two self-adaptive crossover operators for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 89–110. MIT Press, 1996.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

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

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

[5] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187, 1985.

[6] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, 1989.

[7] John R. Koza. *Genetic programming: on the programming of computers by means of natural selection.* MIT Press, 1992.

[8] John R. Koza. *Genetic programming II: automatic discovery of reusable programs.* MIT Press, 1994.

[9] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. *Genetic programming III: darwinian invention and problem solving.* Morgan Kaufmann, 1999.

[10] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic programming IV: routine human-competitive machine intelligence.* Springer, 2003.

[11] John R. Koza and Riccardo Poli. *Introductory Tutorials in Optimization, Search and Decision Support*, chapter 8: A Genetic Programming Tutorial. 2003.

[12] John R. Koza and Riccardo Poli. *Search Methodologies*, chapter 5: Genetic Programming; John Koza and Riccardo Poli. 2005.

[13] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[14] T. Perkis. Stack-based genetic programming. In *Proc IEEE Congress on Computational Intelligence*, 1994.

[15] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Programmers Guide to Genetic Programming.* Lulu Enterprises, 2008.