Clever Algorithms: Programming Paradigms*

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

December 05, 2010
Technical Report: CA-TR-20101205-1

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 considers the use of a range of different programming paradigms that may be used when realizing an algorithm as an implementation.

Keywords: `Clever, Algorithms, Programming, Paradigms, Procedural, Object-Oriented, Flow`

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 [2]. 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 [6]. This report discusses three standard programming paradigms that may be used to implement the algorithms described in the Clever Algorithms Project:

- Procedural Programming (Section 2)
- Object-Oriented Programming (Section 3)
- Flow Programming (Section 4)

Each paradigm is described and an example implementation is provided using the Genetic Algorithm as a context.

2 Procedural Programming

All algorithms in the Clever Algorithms project were implemented using a procedural programming paradigm in the Ruby Programming Language [5]. A procedural representation was chosen to provide the most transferrable instantiation of the algorithm implementations. Many languages support the procedural paradigm and procedural code examples are expected to be easily ported to popular paradigms such as object-oriented and functional.

2.1 Description

The procedural programming paradigm (also called imperative programming) is concerned with defining a linear procedure or sequence of programming statements. A key feature of the paradigm is the partitioning of functionality into small discrete re-usable modules called procedures (subroutines or functions) that act like small programs themselves with their own scope, inputs and outputs. A procedural code example is executed from a single point of control or entry point which calls out into declared procedures, which in turn may call other procedures.

Procedural programming was the first so-called 'high-level programming paradigm' compared to lower level machine code and is the most common and well understood form of programming. Newer paradigms (such as Object-Oriented) and modern businesses programming languages (such as C++, Java and C#) are built on the principles of procedural programming.

2.2 Example

Listing 1 provides an example of the Genetic Algorithm implemented in the Ruby Programming Language using the procedural programming paradigm (taken from [3]). The demonstration problem is a maximizing binary optimization problem called OneMax that seeks a binary string of unity (all '1' bits). The objective function provides only an indication of the number of correct bits in a candidate string, not the positions of the correct bits. The Genetic Algorithm is implemented with a conservative configuration including binary tournament selection for the selection operator, uniform crossover for the recombination operator, and point mutations for the mutation operator.

```
def onemax(bitstring)
  sum = 0
  bitstring.each_char {|x| sum+=1 if x=='1'}
  return sum
end

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

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, prob_mutation)
  child = ""
   bitstring.each_char do |bit|
    child << ((rand()<prob_mutation) ? ((bit=='1') ? "0" : "1") : bit)
  end
  return child
end

def uniform_crossover(parent1, parent2, p_crossover)
  return ""+parent1 if rand()>=p_crossover
  child = ""
```

```ruby
  parent1.length.times do |i|
    child << ((rand()<0.5) ? parent1[i].chr : parent2[i].chr)
  end
  return child
end

def reproduce(selected, population_size, p_crossover, p_mutation)
  children = []
  selected.each_with_index do |p1, i|
    p2 = (i.even?) ? selected[i+1] : selected[i-1]
    child = {}
    child[:bitstring] = uniform_crossover(p1[:bitstring], p2[:bitstring], p_crossover)
    child[:bitstring] = point_mutation(child[:bitstring], p_mutation)
    children << child
    break if children.size >= population_size
  end
  return children
end

def search(max_generations, num_bits, population_size, p_crossover, p_mutation)
  population = Array.new(population_size) do |i|
    {:bitstring=>random_bitstring(num_bits)}
  end
  population.each{|c| c[:fitness] = onemax(c[:bitstring])}
  best = population.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
  max_generations.times do |gen|
    selected = Array.new(population_size){|i| binary_tournament(population)}
    children = reproduce(selected, population_size, p_crossover, p_mutation)
    children.each{|c| c[:fitness] = onemax(c[:bitstring])}
    children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
    best = children.first if children.first[:fitness] >= best[:fitness]
    population = children
    puts " > gen #{gen}, best: #{best[:fitness]}, #{best[:bitstring]}"
    break if best[:fitness] == num_bits
  end
  return best
end

if __FILE__ == $0
  # problem configuration
  num_bits = 64
  # algorithm configuration
  max_generations = 100
  population_size = 100
  p_crossover = 0.98
  p_mutation = 1.0/num_bits
  # execute the algorithm
  best = search(max_generations, num_bits, population_size, p_crossover, p_mutation)
  puts "done! Solution: f=#{best[:fitness]}, s=#{best[:bitstring]}"
end
```

Listing 1: Genetic Algorithm in the Ruby Programming Language using the Procedural Programming Paradigm

3 Object-Oriented Programming

This section considers the implementation of algorithms from the Clever Algorithms project in the Object-Oriented Programming Paradigm.

3.1 Description

The Object-Oriented Programming (OOP) paradigm is concerned with modeling problems in terms of entities called objects that have attributes and behaviors (data and methods) that may interact with other entities via message passing (calling methods on other entities). An object defines a class or template for the entity, which is instantiated or constructed and then may be used in the program.

Objects can extend other objects, inheriting some or all of the attributes and behaviors from the parent providing specific modular reuse. Objects can be treated as a parent type (an object in its inheritance tree) allowing the use or application of the objects in the program without the caller knowing the specifics of the behavior or data inside the object. This general property is called polymorphism, which exploits the encapsulation of attributes and behavior within objects and their capability of being treated (viewed or interacted with) as a parent type.

Organizing functionality into objects allows for additional constructs such as abstract types where functionality is only partially defined and must be completed by descendant objects, overriding where descending objects re-define behavior defined in a parent object, and static classes and behaviors where behavior is executed on the object template rather than the object instance. For more information on Object-Oriented programming and software design refer to a good text book on the subject, such as Booch [1] or Meyer [8].

There are common ways of solving discrete problems using object-oriented programs called patterns. They are organizations of behavior and data that have been abstracted and presented as a solution or idiom for a class of problem. The Strategy Pattern is an object-oriented pattern that is suited to implementing an algorithm. This pattern is intended to encapsulate the behavior of an algorithm as a strategy object where different strategies can be used interchangeably on a given context or problem domain. This strategy can be useful in situations where the performance or capability of a range of different techniques needs to be assessed on a given problem (such as algorithm racing or bake-off's). Additionally, the problem or context can also be modelled as an interchangeable object, allowing both algorithms and problems to be used interchangeably. This method is used in Object-Oriented algorithm frameworks. For more information on the strategy pattern or object-oriented design patterns in general, refer to the so-called 'gang-of-four' design patterns book [7].

3.2 Example

Listing 2 provides an example of the Genetic Algorithm implemented in the Ruby Programming Language using the Object-Oriented Programming Paradigm.

The implementation provides a general problem and strategy classes that define their behavioral expectations. A `OneMax` problem class is defined as is a `GeneticAlgorithm` strategy class. The algorithm makes few assumptions of the problem other than it can assess candidate solutions and indicate the number of bits a candidate solution requires. The problem makes very few assumptions about candidate solutions other than they are hash maps that contain a binary string and fitness key-value pairs. The use of the Object-Oriented strategy pattern means that a new algorithm could easily be defined to work with the defined problem, and that new problems could be defined for the Genetic Algorithm to execute.

Note that Ruby does not support abstract classes, so this construct is simulated by defining methods that raise an exception if they are not overridden by descendant classes.

```ruby
# A problem template
class Problem
  def assess(candidate_solution)
    raise "A problem has not been defined"
  end

  def is_optimal?(candidate_solution)
```

```ruby
    raise "A problem has not been defined"
  end
end

# An strategy template
class Strategy
  def execute(problem)
    raise "A strategy has not been defined!"
  end
end

# An implementation of the OneMax problem using the problem template
class OneMax < Problem

  attr_reader :num_bits

  def initialize(num_bits=64)
    @num_bits = num_bits
  end

  def assess(candidate_solution)
    if candidate_solution[:bitstring].length != @num_bits
      rase "Expected #{@num_bits} in candidate solution."
    end
    sum = 0
    candidate_solution[:bitstring].each_char {|x| sum+=1 if x=='1'}
    return sum
  end

  def is_optimal?(candidate_solution)
    return candidate_solution[:fitness] == @num_bits
  end
end

# An implementation of the Genetic algorithm using the strategy template
class GeneticAlgorithm < Problem

  attr_reader :max_generations, :population_size, :p_crossover, :p_mutation

  def initialize(max_gens=100, pop_size=100, crossover=0.98, mutation=1.0/64.0)
    @max_generations = max_gens
    @population_size = pop_size
    @p_crossover = crossover
    @p_mutation = mutation
  end

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

  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)
    child = ""
     bitstring.each_char do |bit|
      child << ((rand()<@p_mutation) ? ((bit=='1') ? "0" : "1") : bit)
    end
    return child
  end

```

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

  def reproduce(selected)
    children = []
    selected.each_with_index do |p1, i|
      p2 = (i.even?) ? selected[i+1] : selected[i-1]
      child = {}
      child[:bitstring] = uniform_crossover(p1[:bitstring], p2[:bitstring])
      child[:bitstring] = point_mutation(child[:bitstring])
      children << child
      break if children.size >= @population_size
    end
    return children
  end

  def execute(problem)
    population = Array.new(@population_size) do |i|
      {:bitstring=>random_bitstring(problem.num_bits)}
    end
    population.each{|c| c[:fitness] = problem.assess(c)}
    best = population.sort{|x,y| y[:fitness] <=> x[:fitness]}.first
    @max_generations.times do |gen|
      selected = Array.new(population_size){|i| binary_tournament(population)}
      children = reproduce(selected)
      children.each{|c| c[:fitness] = problem.assess(c)}
      children.sort!{|x,y| y[:fitness] <=> x[:fitness]}
      best = children.first if children.first[:fitness] >= best[:fitness]
      population = children
      puts " > gen #{gen}, best: #{best[:fitness]}, #{best[:bitstring]}"
      break if problem.is_optimal?(best)
    end
    return best
  end
end

if __FILE__ == $0
  # problem configuration
  problem = OneMax.new
  # algorithm configuration
  strategy = GeneticAlgorithm.new
  # execute the algorithm
  best = strategy.execute(problem)
  puts "done! Solution: f=#{best[:fitness]}, s=#{best[:bitstring]}"
end
```

Listing 2: Genetic Algorithm in the Ruby Programming Language using the Object-Oriented Programming Paradigm

4 Flow Programming

This section considers the implementation of algorithms from the Clever Algorithms project in the Flow Programming Paradigm.

4.1 Description

Flow, data-flow, or pipeline programming involves chaining a sequence of smaller processes together and allowing a flow of information through the sequence in order to perform the desired computation. Units in the flow are considered black-boxes that communicate with each other via message passing. The information that is passed between the units is considered a stream and a given application may have one or more streams of potentially varying direction. Discrete information in a stream is partitioned into information packets which are passed from unit-to-unit via message buffers, queues or similar data structures.

A flow organization of functionality allows computing units to be interchanged readily with variations. It also allows for variations of the pipeline to be considered with minor reconfiguration. A flow or pipelining organization is commonly used by algorithm frameworks for the organization within a given algorithm implementation, allowing the specification of operators that manipulate the flow of candidate solutions to be varied and interchanged.

For more information on Flow-based programming see a good textbook on the subject, such as Morrison [9].

4.2 Example

Listing 3 provides an example of the Genetic Algorithm implemented in the Ruby Programming Language using the Flow Programming Paradigm. Each unit is implemented as an object that executes its logic within a standalone thread that forever will attempt to read input from the input queue and write data to the output queue. The implementation shows four flow units organized into a cyclic graph where the output message queue of one unit is used as the input message queue of the next unit in the directional cycle (`EvalFlowUnit` to `StopConditionUnit` to `SelectFlowUnit` to `VariationFlowUnit`).

Candidate solutions are the unit of data that is passed around in the flow between units. The system is started although does not have any information to process until a set of random solutions are injected into the evaluation unit's input queue. The solution are evaluated and sent to the stop condition unit where the constraints of the algorithm execution are tested (optima found or max number of evaluations) and the candidates are passed on to the selection flow unit. The selection unit collects a fixed number of candidate solutions then fitness-proportionally selects candidate solutions that are passed onto the variation unit. The variation unit performs crossover and mutation on each pair of candidate solutions then sends the results to the evaluation unit, completing the cycle.

```
require 'thread'

# Generic flow unit
class FlowUnit
  attr_reader :queue_in, :queue_out, :thread

  def initialize(q_in=Queue.new, q_out=Queue.new)
    @queue_in, @queue_out = q_in, q_out
    start()
  end

  def run
    raise "FlowUnit not defined!"
  end

  def start
    puts "Starting flow unit: #{self.class.name}!"
    @thread = Thread.new do
      run() while true
    end
  end
```

```ruby
end

# Evaluation of solutions flow unit
class EvalFlowUnit < FlowUnit
  def onemax(bitstring)
    sum = 0
    bitstring.each_char {|x| sum+=1 if x=='1'}
    return sum
  end

  def run
    data = @queue_in.pop
    data[:fitness] = onemax(data[:bitstring])
    @queue_out.push(data)
  end
end

# Stop condition flow unit
class StopConditionUnit < FlowUnit
  attr_reader :best, :num_bits, :max_evaluations, :evals

  def initialize(q_in=Queue.new, q_out=Queue.new, max_evaluations=10000,num_bits=64)
    super(q_in, q_out)
    @best, @evals = nil, 0
    @num_bits = num_bits
    @max_evaluations = max_evaluations
  end

  def run
    data = @queue_in.pop
    if @best.nil? or data[:fitness] > @best[:fitness]
      @best = data
      puts " >new best: #{@best[:fitness]}, #{@best[:bitstring]}"
    end
    @evals += 1
    if @best[:fitness]==@num_bits or @evals>=@max_evaluations
      puts "done! Solution: f=#{@best[:fitness]}, s=#{@best[:bitstring]}"
      @thread.exit()
    end
    @queue_out.push(data)
  end
end

# Fitness-based selection flow unit
class SelectFlowUnit < FlowUnit
  def initialize(q_in=Queue.new, q_out=Queue.new, pop_size=100)
    super(q_in, q_out)
    @pop_size = 100
  end

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

  def run
    population = Array.new
    population << @queue_in.pop while population.size < 100
    @pop_size.times do
      @queue_out.push(binary_tournament(population))
    end
  end
end
```

```ruby

# Variation flow unit
class VariationFlowUnit < FlowUnit
  def initialize(q_in=Queue.new, q_out=Queue.new, crossover=0.98, mutation=1.0/64.0)
    super(q_in, q_out)
    @p_crossover = crossover
    @p_mutation = mutation
  end

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

  def point_mutation(bitstring)
    child = ""
     bitstring.each_char do |bit|
      child << ((rand()<@p_mutation) ? ((bit=='1') ? "0" : "1") : bit)
    end
    return child
  end

  def reproduce(p1, p2)
    child = {}
    child[:bitstring] = uniform_crossover(p1[:bitstring], p2[:bitstring])
    child[:bitstring] = point_mutation(child[:bitstring])
    return child
  end

  def run
    parent1 = @queue_in.pop
    parent2 = @queue_in.pop
    @queue_out.push(reproduce(parent1, parent2))
    @queue_out.push(reproduce(parent2, parent1))
  end
end

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

if __FILE__ == $0
  # create the pipeline
  eval = EvalFlowUnit.new
  stopcondition = StopConditionUnit.new(eval.queue_out)
  select = SelectFlowUnit.new(stopcondition.queue_out)
  variation = VariationFlowUnit.new(select.queue_out, eval.queue_in)
  # push random solutions into the pipeline
  100.times do
    solution = {:bitstring=>random_bitstring(64)}
    eval.queue_in.push(solution)
  end
  stopcondition.thread.join
end
```

Listing 3: Genetic Algorithm in the Ruby Programming Language using the Flow Programming Paradigm

5 Other Paradigms

A number of popular and common programming paradigms have been considered in this report, although many more have not been described.

Many programming paradigms are not appropriate for implementing the algorithm as-is, but may be useful with the algorithm as a component in a broader system, such as Agent-Oriented Programming where the algorithm may be a procedure available to the agent. Meta-programming is another interesting case where capabilities of the paradigm may be used for parts of an algorithm implementation, such as the manipulation of candidates program in algorithms such as Genetic Programming [4].

Other programming paradigms provide variations on what has already been described, such as Functional Programming which would not be too dissimilar to the procedural example, and Event-Driven Programming that would not be too dissimilar in principle to the Flow-Based Programming. Another example is the popular idioms such as the Map-Reduce paradigm is an application of functional programming principles organized into a data flow model.

Finally, there are programming paradigms that are not relevant or feasible to consider implementing such algorithms, for example Logic Programming, Declarative Programming and Aspect-Oriented Programming.

6 Conclusions

This report considered the implementation of Clever Algorithms in a range of common and popular programming languages.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, 1997.

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

[3] Jason Brownlee. Genetic algorithm. Technical Report CA-TR-20100303-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, March 2010.

[4] Jason Brownlee. Genetic programming. Technical Report CA-TR-20100308-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, March 2010.

[5] Jason Brownlee. Programming language selection for optimization algorithms. Technical Report CA-TR-20100122-1, The Clever Algorithms Project http://www.CleverAlgorithms.com, January 2010.

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

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.

[8] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[9] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development.* CreateSpace, 2nd edition edition, 2010.